António Miguel Ribeiro dos Santos Rosado da Cruz

Automatic Generation of User Interfaces from Rigorous Domain and Use Case Models

Departamento de Engenharia Informática
Faculdade de Engenharia da Universidade do Porto

September 17, 2010

António Miguel Ribeiro dos Santos Rosado da Cruz

Automatic Generation of User Interfaces from Rigorous Domain and Use Case Models

Dissertation submited for the degree of
Doctor of Philosophy in Informatics Engineering

Dissertation conducted under the scientific supervision of
Prof. Dr. João Carlos Pascoal Faria
Assistant Professor, Department of Informatics Engineering, FEUP

September 2010

Abstract

User interface (UI) development, in the scope of data intensive interactive applications, is a time consuming but repetitive task. Nevertheless, few research projects address its automatic generation.

Existing model-driven approaches to the UI automatic generation either demand the full construction of a UI model, which, with data intensive applications, corresponds to moving the repetitiveness of the UI construction to the modeling level, or demand a set of complex sub-models polluted with concrete UI information. This situation sets aside a more generalized utilization of such approaches. A few solutions found in the literature try to simplify the demanded sub-models by generating other sub-models from the previous ones, but they have revealed to be very inflexible, making it hard to work around less "standard" problems.

Based on the identification and comparison of the state-of-art tools and approaches to the automatic generation of user interfaces, this Ph.D. research work addresses the automatic generation of data driven interactive applications, including its user interface, following a model-driven paradigm. The proposed approach starts from platform independent non-UI models of the system under development, namely its domain model and its use case model, and generates a UI model, which is used, together with the domain model and use case model, to generate the completely functional final code, which may be used as a prototype or as a step for posterior refinements towards the final application.

An iterative development process for data intensive interactive applications, aligned with the model-driven architecture (MDA), is also addressed, comprising model validation through a generated prototype at the end of each iteration. The presented approach shall be viewed in an evolutionary development perspective, starting with a prototype that enables the validation and execution of executable system models, in an early phase of the software development process, and being possible to use it as a base for subsequent developments, by refining the previous models or complementing them with new sub-models.

OCL and an action semantics language are used to add rigor and semantic richness to the system model, and allow the generation of features derived, for instance, from the operations' body, invariants and preconditions defined in the model, that contribute to the enhancement of the UI usability and acceptability.

Two case studies are presented to validate the proposed approach.

Résumé

Le développement d'interfaces utilisateur (IU) dans le cadre d'applications interactives intensivement basées sur des données, est une tâche de longue haleine, mais répétitif. Néanmoins, peu de projets de recherche visent leur génération automatique.

Les approches existantes pour la génération automatique d'interfaces utilisateur, ou exigent la construction d'un modèle complet de l'interface utilisateur, qui, dans les applications intensivement basées sur des données, il signifie de déplacer la répétabilité de la construction de l'IU pour le niveau de la modélisation, ou exigent un ensemble complexe de sous-modèles contaminés par d'information spécifique de l'IU. Cette situation empêche une plus large utilisation de ces approches. Certaines solutions existantes dans la littérature, essayent la simplification des sous-modèles requis par la production d'autres sous-modèles à partir des précédents, mais ces solutions ont révélé être très rigides, rendant difficile de travailler autour de problèmes moins "standard".

Fondée sur l'identification et la comparaison des outils et des approches de l'état de l'art à la génération automatique des interfaces utilisateur, ce travail de recherche se concentre sur la génération automatique des applications interactives intensivement basées sur des données, y compris son interface utilisateur, suivant un paradigme dirigé par les modèles. L'approche commence par la modélisation du système en cours de développement, non de l'interface utilisateur, indépendamment de la plateforme, à savoir le modèle de domaine et le modèle de cas d'utilisation, et génère un modèle de l'interface utilisateur, qui est utilisé avec le modèle de domaine pour générer le code de l'application finale.

Il est également proposé un processus itératif de développement de logiciels, aligné avec le Model Driven Architecture (MDA), y compris la validation des modèles en utilisant un prototype fonctionnel à la fin de chaque itération. L'approche présentée doit être contextualisée dans une perspective évolutive du développement de logiciels, à partir d'un prototype qui permet la validation et la mise en uvre de modèles exécutables du système, et permettre leur utilisation comme base pour le développement futur grâce à l'amélioration des les modèles précédents, ou de compléter avec de nouveaux sous-modèles.

L'approche proposée utilise le langage OCL (Object Constraint Language) et un langage d'actions pour accroître la rigueur et la richesse sémantique du modèle du système en cours de développement, permettant la génération de fonctionnalité provenant des corps des opérations, invariants et pré-conditions définies dans le modèle, en contribuant à améliorer la convivialité et l'acceptabilité de l'interface utilisateur.

Sont présentés Deux études de cas qui valident l'approche proposée dans cette recherche.

Resumo

O desenvolvimento de interfaces com o utilizador (IU), no âmbito de aplicações interactivas intensivamente baseadas em dados, é uma tarefa repetitiva e grande consumidora de tempo. No entanto, poucos projectos de investigação têm como objectivo a sua geração automática.

As abordagens existentes para geração automática de interfaces com o utilizador, ou exigem a construção completa de um modelo da IU, o que, em aplicações intensivamente baseadas em dados, corresponde a transportar a repetitividade da construção da IU para o nível da modelação, ou exigem um conjunto complexo de sub-modelos poluídos com informação concreta da IU. Esta situação coloca de lado uma utilização mais generalizada de tais abordagens. Algumas soluções, encontradas na literatura, tentam simplificar os sub-modelos exigidos através da geração de outros sub-modelos a partir dos primeiros, mas essas soluções revelaram ser pouco flexíveis tornando complicado o tratamento de problemas menos padronizados.

Baseado na identificação e comparação das ferramentas e abordagens do estado da arte, para geração automática de IU, este trabalho de investigação aborda a geração automática de aplicações interactivas completamente funcionais intensamente baseadas em dados, incluindo a sua IU, seguindo uma abordagem guiada por modelos. A abordagem defendida começa com a construção de modelos do sistema em desenvolvimento, não da IU, independentes da plataforma, nomeadamente modelo de domínio e modelo de casos de uso, e gera um modelo da IU, o qual é usado juntamente com o modelo de domínio para gerar o código final da aplicação.

É também proposto um processo iterativo, de desenvolvimento de software, alinhado com a arquitectura guiada por modelos (MDA), compreendendo a validação de modelos usando um protótipo funcional, no final de cada iteração. A abordagem apresentada deve ser contextualizada numa perspectiva de desenvolvimento de software evolucionária, começando com um protótipo que permite a validação e execução de modelos executáveis, do sistema, numa fase inicial do processo de desenvolvimento de software, e tornando possível a sua utilização como base para futuros desenvolvimentos, através do refinamento dos modelos anteriores, ou do seu complemento com novos sub-modelos.

É usada a linguagem OCL (Object Constraint Language) e uma linguagem de acções para adicionar rigor e riqueza semântica ao modelo do sistema em desenvolvimento, permitindo a geração de algumas funcionalidades derivadas dos corpos das operações, invariantes e pré-condições definidas no modelo, contribuindo para a melhoria da usabilidade e aceitabilidade da IU.

São apresentados dois casos de estudo que validam a abordagem proposta neste trabalho de investigação.

Acronyms

AS Action Semantics

CAP Canonical Abstract Prototypes

CIM Computation Independent Model

CLI Command Line Interface

CSS Cascading Style Sheets

CTT ConcurTaskTrees

DM Domain Model

DSL Domain Specific Language

DSM Domain Specific Modeling

GOMS Goals, Operators, Methods and Selection Rules

GUI Graphical User Interface

HCI Human-Computer Interaction

LHS Left-hand side

M2C Model-to-code transformation

M2M Model-to-model transformation

M2T Model-to-text transformation

MB-UIDE Model-based User Interface Development Environment

MDA Model Driven Architecture

MDD Model Driven Development

MDSD Model Driven Software Development

MOF Meta Object Facility

OCL Object Constraint Language

OMG Object Management Group

OO Object oriented

PIM Platform Independent Model

PSM Platform Specific Model

QVT Query/View/Transform

RDF Resource Description Framework

RHS Right-hand side

SE Software Engineering

UCM Use Case Model

UI User Interface

UIDL User Interface Description Language

UIM User Interface Model

UML Unified Modeling Language

UP Unified Process

URI Uniform Resource Identifier

WIMP Windows, Icons, Menus and Pointing devices

XML eXtensible Markup Language

XSLT eXtensible Stylesheet Language

XUL XML User Interface Language

Contents

List of Figures

List of Tables

Acknowledgments

Having decided to follow a Ph.D. program, after changing my career from the enterprise to the academic world, I contacted Prof. Dr. João Pascoal Faria, whom I already knew from joint works in the software industry. He readily agreed to be my Ph.D. supervisor. Four years have passed, since the inception of this Ph.D. program, and I would like to greatly thank Prof. Pascoal Faria for his guidance, enthusiastic discussions, stimulating supervision and firm support.

I also want to thank the members of my reading committee, in alphabetical order, Prof. Dr. Alberto Silva, from the Technical Higher Institute of the Technical University of Lisbon (IST), and Prof. Dr. Ana Paiva, from the Faculty of Engineering of the University of Porto (FEUP).

I would also like to thank Prof. Dr. Rui da Silva Gomes, from the Polytechnic Institute of Viana do Castelo (IPVC), where I work since 2005, for all the encouragement in pursuing a Ph.D., and for all the efforts in facilitating the management of my time between the Ph.D. project and the lectures at the IPVC.

I also want to thank João Saraiva and David Ferreira, PhD students at IST, and Filipe Correia, Rui Gomes, Hugo Sereno and André Restivo, PhD students at FEUP, for their help either in facilitating the access to MDD tools they were working on, or by providing comments about articles I have submitted to conferences.

These acknowledgements wouldn't be complete without mentioning my family. A very special thanks goes to my wife Estrela and to my son André, for their support, infinite patience, and tolerance, during my countless periods of "absence", when I stayed "cloistered" at home working for this dissertation.

A special thanks goes also to my parents, Maria do Carmo and António Joaquim, for always having invested in their children's education and for having taught me that studying and working hard is the most rewarding way of meeting life's goals.

Finally, I would also want to remember my grandparents, Vitor, Etelvina, Joaquim and Esperança, because they are always popping into my mind bringing me their teachings by making me remember past conversations and life examples.

To Estrela and André

To my parents

To the memory of my grand-parents

Chapter 1

Introduction

1.1 Context and Motivation

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. A 1992 survey [MR92] concluded that user interface (UI) development time was about 50% of the total development time for a software system's project. This ratio is probably close to today's reality as, since then, there are better integrated software development environments that enable the visual development of graphical user interfaces (GUI) and, on the other side, applications' UI is much more complex since the advent of Web applications. There are many reasons for this UI weight in the software development process. UI deals with the user's intentions and the system's responsibilities. This is a very complex task and, for that reason, software engineers develop modeling or prototyping activities. A prototype facilitates the communication with the stakeholders, especially with the end users, and allows for the validation of elicited requirements and of design decisions.

1.1.1 User interface development

Several approaches exist for user interface development. [Mar07] considers the following types of tools for developing graphical user interfaces:

- **Sketching tools** - enable the graphical design of user interfaces, by making possible to draw the UI, or by recognizing handwritten sketches (e.g.: SILK [Lan96], DENIM [LNHL02]).

- **UI builders** - continue to be the most popular and productive UI development tools. These enable the graphical construction of UI concrete

artifacts, by dragging and dropping UI widgets, and are usually fully integrated with the development environment of the code behind the UI (e.g.: Microsoft Visual Studio [MSDa], Sun Net Beans [Sun]).

- **XML-based UI description languages** - enable the description of user interfaces in a platform independent manner (e.g.: XIML [PE01], XUL [Moz], XAML [MSDb]), by using a user interface description language (UIDL) based on XML.
- **Model-based UI approaches** - enable the development of UIs by constructing models composed of different sub-models, each modeling a specific aspect of the system (e.g.: Mastermind [SSC+96]).

1.1.2 Modeling and Prototyping

Modeling is a well established way people take for dealing with complexity. A model allows one to focus on important properties of the system being modeled (e.g., a house, a car, a piece of software) 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.

Modeling interactive computer systems involves the separate modeling of different system's concerns. For each of these concerns, a different artifact, or sub-model, is typically developed. Each sub-model describes a different viewpoint of the model. All these viewpoints shall be consistent with each other because they are all views of the same model, and describe the same system. When the models serve the sole purpose of helping humans to understand the reality (the system being modeled), some incompleteness or ambiguity may be allowed. But, if the models are to be processed automatically, by computers, in a model transformation or a code generation process, then ambiguity or incompleteness cannot be accepted.

An important concern to be modeled during an interactive system design is the "core" system's structure and functionality. That is the system domain entities and the functionality provided by each entity class for collaborating with other classes and fulfill system responsibilities.

The user interface is another concern that can be modelled during system design. UI models address the translation between what the user wants and what the system does [DFAB98]. UI models are concerned with the UI functionality or tasks provided to the user, its structure, its presentation to the user, and its usability.

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 leverage 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, the development of user task analysis for modeling the user intended tasks on the interactive system is one of the concerns that shall be modeled. Typically, task analysis and modeling involve the development of goal and task hierarchies and the identification of objects and actions involved in each task [DFAB98]. Besides this task model development, a view of the UI relevant aspects of the system core structure and functionality may also be modeled, along with a UI presentation model, in order to complete the whole interactive system model.

In the SE community, a common software engineering practice is to build a UML system model, comprising a domain model and a use case model, supplemented by a non-functional user interface prototype, in the early stages of the software development process [JBR99, Pre05]. The domain model enables a structural view of the system, and 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 viewpoint through UML use case diagrams and accompanying textual descriptions.

The non-functional user interface prototype is used to elicit and validate requirements with end users and customers. It may help eliciting ambiguous or complex requirements - by letting end users explain how they'd see themselves using the system for fulfilling their goals -, in which case a throw-away prototype is good enough, or may be used initially for helping validate the system model with the end users - in order to "synchronize" the end users' viewpoints of the system with the team members' points of view -, and may then be refined through a number of stages to the final system, in which an evolutionary prototype is obviously needed [Som07].

Typically, the user interface prototype is not integrated with the system model, that is the user interface prototype cannot be used to interact with an executable model.

The use case and domain models are typically ambiguous and incomplete, and its consistency cannot be automatically validated, because most of the constraints are specified in textual natural language. This kind of models is mainly used to reason about the system being built, and for sharing information between the project team members and other stakeholders.

If these model views are to be used in an automatic generation process, or a model transformation process, like predicted by model-driven software development approaches, then they shall be consistent, unambiguous (rigorous), and complete. If a system model is completely and rigorously specified, then it may be executed, either by simulation, if a proper virtual machine is provided, or by generating executable code directly from the model, in one or more steps, if

proper code generation tools are provided.

1.1.3 Model-driven Development

Model driven development (MDD) approaches, like Domain Specific Modeling (DSM) [KT08], or the OMG's Model Driven Architecture (MDA) [WBP+03], 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. Model driven development is mainly focused on platform independent modeling activities rather than programming activities. This allows software engineers to focus on concepts of the problem domain, and the way they shall be modeled in order to produce a software solution, rather than being distracted by technical issues of the solution domain. Within an MDD setting, code can be automatically generated to a great extension, dramatically reducing the most costly and error-prone aspects of software development [Fra03].

These concepts will be further discussed in chapter 2.

Today UML-based MDD is typically used to produce only part of the solution, namely the database creation scripts, and archetypes for the modeled classes, relations and methods. This is partially because software engineers don't use the full power of UML to accomplish a complete and rigorous modeling activity. In fact, models are often incomplete, because the modeled parts of the system are only the ones that can be modeled diagrammatically. Another reason is that UML itself has some difficulties in providing an easy and full integration between different model views. This adds to the fact that almost none of the industry tools, that permit the modeling and transformation of models within an MDD setting, provide full support both for UML and for model transformations. Despite that, the standard UML includes the definition of a constraint specification language, namely OCL (Object Constraint Language), and a procedural actions language, namely Action Semantics, that allow the unambiguous and complete modeling of a system, but are seldom used by modelers.

Chapter 3 revises the models used in model-based UI development, and surveys the state of the art tools and approaches for model-driven UI code generation.

1.2 Limitations of current approaches

The model-driven development of interactive applications demands the construction of several model views, as discussed in chapter 2, usually including a User Interface Model (UIM). Current approaches to the UI automatic code generation from models, surveyed in chapter 3, have the following limitations:

- In general, current approaches demand too much effort, from the modeler, in order to build the system models required as input by those approaches. They don't allow a gradual approach to system modeling if one wants to

generate a (prototype) application to iteratively evaluate and refine the model. Almost all models expected by one approach must be fully developed before code generation may be available.

- 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.
- 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. Some approaches allow the interactive modeling and animation of use case driven UML models, but they are based on the direct animation of the models in a simulation environment, not in the animation through a derived UI, so it may not be used by users or other stakeholders than the modeling team members themselves.
- Most of the existing approaches don't allow a complete model specification that allows thw derivation of validation features, for example by taking advantage of the specification of class' state constraints (invariants) or of operations preconditions to enhance the usability of the generated UI.
- Existing approaches don't allow a flexible use case specification, for instance by taking advantage of the use case relations or of constructs typically found in task models [PMM97, Pat03] (e.g.: sequencing, alternative) for detailing use cases.
- Existing approaches don't allow the complete definition of operations at class level, for example by providing an action language that would enable the definition of the semantic of operations.
- Existing approaches, even the ones that generate CRUD operations by default, don't allow the modification of the standard CRUD behavior, for example by enabling the definition of triggers. Triggers are operations triggered by the invocation of other operation or by the holding of a given condition. Triggers activated by an operation's invocation are a way of modifying or adding behavior to CRUD or other operations. By using triggers it is possible to easily specify business rules that involve several classes' operations.

1.3 Research Questions

A set of research questions has been established based on the conclusions from the state of the art discussion (see section 1.2, further details in chapter 3). The

research questions formulated in order to respond to the identified problems of current approaches, are:

1. Is it possible to obtain a UI prototype from a minimal set of model artifacts (a rigorous domain model and a rigorous use case model), without requiring as input a UI model?

2. Can the generated UI prototype take advantage of advanced elements in a domain model:

   - state invariants;
   - operations' pre-/post-conditions;
   - operations rigorously and completely defined by the modeler, that is operations which are completely and formally specified and for which there is only one interpretation;
   - derived attributes;
   - derived classes;
   - triggers or other means of modifying standard CRUD behavior.

3. Can the generated UI prototype take advantage of more flexible elements, in a use case model:

   - typical use case relations (e.g.: inclusion, extension, inheritance);
   - use of constructs typically found in task models for detailing use cases.

4. Can the generated UI be used as a starting point for further refinements towards the UI of the final application?

The answers to these questions are given in section 6.7.

1.4 Research Goal

The goal of this PhD research work is to improve current approaches to model-driven automatic UI generation, namely UIM derivation from system non-UI platform independent models, addressing all issues identified as flaws in the current approaches. The derived UI model, together with the precisely constructed early system model, shall allow the early validation of the system model and help in requirements elicitation and validation. The UI generated can be subject to usability and appearance improvements (without loosing the links to the underlying system model), and can also be used as a basis for subsequent system development.

Further iterations of modeling and prototype usage (with user feedback) shall enable the refinement of the system model and its enrichment with more model

elements. For enhancing the preciseness of the model, OCL predicates are used to formalize domain class invariants and domain classes operations' preconditions.

The UI generation process takes advantage of the OCL invariants and of the operations' pre-conditions, defined in the domain model, and of use case model features, to enhance the usability and behavior of the UI.

User defined operations' body is fully specified by making use of an actions language, in order to guarantee the executability of the generated prototype.

The developed approach is focused on business applications and form based UIs.

We are now in position to state the thesis of this PhD research work.

Thesis:

We claim that it is possible to automatically generate a usable user interface from early, semantically rich, system models, demanding from the modeler less effort than the existing approaches. The generated default user interface may be *tunned* by a UI designer in two points of the process: - After having generated an abstract UI model, but before generating a concrete UI; and, after generating a concrete UI in a XML-based UI description language, that allows for the *a posteriori* customization and application of style sheets.

Furthermore, we believe that it is possible to generate different user interfaces, depending on the degree of refinement of the information that the user includes in the model. This will enable the generation of simple user interfaces from simple early structural models, and complex user interfaces from semantically richer system platform independent models. The early UIs generated from early models, may serve for eliciting complex requirements or test the constructed model by executing it through a UI. The UIs generated from semantically richer models may be used for producing the final application UI.

1.5 Research Method and Techniques

Scientific research involves methods and techniques [Oli09]. Research methods refer to the manner in which a research project is undertaken, and research techniques have to do with the specific means whereby data is gathered and analysed, and how inferences are drawn.

Research methods may be classified, according to the motivation for the research, as: - Pure research; and, - Instrumentalist research. The latter may, in turn, be divided into applied research and problem-oriented research [Oli09, Kah01].

According to the base research theory, research methods may be classified as [Oli09, Gre04]: - Descriptive; - Explicative; - Predictive; and, - Prescriptive.

And, according to the tradition in a given area of knowledge domain, research methods may be classified as [Oli09, LM05, LM06]: - Quantitative; - Qualitative; and, - Engineering-oriented.

The research method used in this research work may be classified as:

1. Classification depending on the motivation:

   - Problem-oriented instrumentalist research: The problem is well defined and the research consists in finding techniques to resolve it. It has also characteristics of an applied research, as the research method also starts with a technology, namely MDD, and applies it to solve the defined problem.

2. Classification depending on the basic research theory:

   - Prescriptive research: Prescribe and apply activities and standards in specific circumstances.

3. Classification depending on the tradition in the knowledge area:

   - Engineering-oriented research: Solve the stated problem making use of technology, through models and/or requirements conceptualization, prototyping, construction, demonstration and, finaly, an assessment phase.

Research techniques used in this research work include [Oli09, SDJ07]:

1. State-of-art revision and analysis, also known as disciplined literature review or systematic review [1];

2. Engineering-based techniques, namely the technique of construction, including the conceptualization, design and prototyping (construction) of an artifact/system to explicitly solve a problem or test an hypothesis.

For assessing, or validating, the research results a set of case studies have been used and a feature-based comparison to the state-of-art approaches has been accomplished. Another privileged means of assessment of the approach was the submission of papers to international and/or national scientific conferences on software engineering and on other software development related issues.

[1] http://www.rogerclarke.com/SOS/DLR.html

1.6 Summary of Contributions

In this research work the development of an approach for the model-driven automatic generation of fully functional (executable) interactive applications, from early system models, with minimum effort, was pursued. The generated application may serve, in a first instance, for model validation and requirements elicitation and validation purposes, and in ulterior iterations, as an approximation to the final application.

The main contributions of this Ph.D. research work are:

- the definition of a process for the automatic generation of user interfaces (model and executable prototype) from domain and use case models, which may be instantiated from any iterative incremental (evolutionary) software development process, namely agile processes, and promotes a model-driven software development methodology with minimal model construction.

- the definition of a metamodel for domain and use case models, that extends the UML metamodel, and better enables taking profit of the model features when generating the UIM;

- the definition of a UI metamodel that allows the platform independent modeling of UI structure and of the bindings from UI structure to the domain model, therefore containing all the information for generating a fully executable prototype;

- a set of model transformation rules that allow the derivation of a default UIM from the Domain Model (DM) and the Use Case Model (UCM).

- the development of a proof of concept tool that supports and validates the conclusions of this research work.

The proposed approach builds on the identified limitations in current approaches, improving the state-of-art as presented in the following list. For each one of the identified limitations, this work's contribution to its mitigation is presented:

- Current approaches demand too much effort, from the modeler, in order to build the system models required as input by those approaches.

  - Our approach enables a gradual approximation to system modeling, by being able to derive a default UI and an executable prototype from the domain model alone or from the domain model and the use case model. It is also possible to have these initial models in different levels of abstraction or rigour, and refine them in an evolutive manner.

- 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.

  - Our approach is able to generate a UI model from the system's non-UI sub-models. This helps the modeler in creating a model for the final application and, thereby, reduces the effort required.

- Current approaches don't allow the generation of an executable prototype from the available system models.

  - Our approach is able to derive a default UI and an executable prototype from the domain model (DM) alone, and from both the DM and the use case model, turning possible to interactively evaluate the system model with the end users, and to iteratively evaluate and refine the model. It also allows to add rigour and model elements to the system model, generating refined UIs and refined executable prototypes that support an evolutive model-driven development with the close participation of end users.

- 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.

  - Our approach takes advantage of class invariants and of operation pre-conditions to generate validation routines in the executable application. This enhances the usability of the generated UI by helping the user in entering valid data into forms, and by giving feedback identifying invalid data.

- Existing approaches don't take advantage of the use of constructs typically found in task models (e.g.: enabling, disabling, choice) for detailing use cases.

  - Our approach looks at use cases as the typical software engineering concept of system packaged functionality, but makes use of use case relations and of constructs typically found in task models [PMM97, Pat03], to detail use cases in a manner closer to the end user's point of view. This approximates the use case model to a task model, which is a well known HCI technique to model user actions on the system [DFAB98].

- Existing approaches don't allow the definition of the semantic of operations at class level.

  - Our approach makes use of an Actions Language to specify the semantic of operations at class level.

- Existing approaches don't allow the definition of triggers.

  - Our approach enables the definition of triggers activated by the invocation of an operation and activated by the holding of a given state condition.

The previously stated contributions were described in the following papers presented in scientific conferences after being approved through a peer reviewing process:

1. 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.

2. Cruz, Antonio M. (2007). Deriving Default User Interfaces From Domain Contracts. In Novas Perspectivas em Sistemas e Tecnologias de Informação, Proceedings of the 2nd Iberian Conference on Information Systems and Technologies (CISTI 2007), vol. II, pp 243-253, University of Fernando Pessoa, Porto, Portugal, June 2007.

3. 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.

4. 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.

5. Cruz, A.M.R., Faria, J.P. (2010). A Metamodel-based approach for automatic User Interface generation. In Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering (Models 2010), Oslo, Norway, October 2010 (to appear).

1.7 Overview of the dissertation

This dissertation is organized into a total of seven chapters.

Chapter 2

This chapter presents some concepts and definitions that are needed throughout the dissertation. It briefly points out model-driven software development and user interface development concepts.

Chapter 3

This chapter presents a state of the art survey on UI automatic generation, and analyses the current approaches through the definition of a comparison framework.

Chapter 4

This chapter presents a model-driven software development process, that may be instantiated from any iterative incremental process, which is based on the automatic generation of the final application from platform independent models. The chapter also presents the UML-aligned metamodels used for developing the system model, and the MOF-based metamodel for creating UI models.

Chapter 5

This chapter presents the mapping rules between metamodel elements that allow the model-to-model transformations that enable the generation of a use case model from a domain model and a user interface model from domain and use case models. The rules are presented together with a running example.

Chapter 6

This chapter presents the implementation of a proof-of-concept tool and two case studies, discusses results obtained, and assesses the satisfaction of the proposed research goals.

Chapter 7

This chapter summarizes the results obtained, presents conclusions and points out open issues for future research.

Chapter 2

Concepts and Definitions

This chapter presents some concepts and definitions that are needed throughout the dissertation. It briefly points out model-driven software development and user interface development concepts.

2.1 Processes and Methods

Software development is typically triggered by the need to solve a problem detected at the information system's level of an organization. This problem may have several origins, such as satisfying an information requirement, streamline business tasks, enhance the performance of people and machines, etc.. These needs are usually compiled in the form of a more or less detailed requirements list. The use of techniques to elicit and model these requirements is usually the first stage in a software development process. A software development process, or software development life cycle, consists of a sequence of activities and associated results that produce a software product [Som07]. Each software development organization or project defines its own software development process, i.e. defines the activities for constructing a software product and the way these are organized [Pre05, Som07, SV05].

Software development processes may be based on, or abstracted by, software process models, such as the waterfall or evolutionary models [Som07, Pre05].

There are several software development process models, which may be combined or used as starting point for a software process definition [Pre05] (e.g. Waterfall, Spiral, Unified Process).

Modern software development processes, like the Unified Process [JBR99] (UP), typically use an iterative and incremental approach for developing software. This allows the software engineers to cope with the ambiguities of the human language used for user requirements elicitation, and mitigate risks as at the end of each iteration the users and other stakeholders are encouraged to give feedback about the software increment that has been delivered to them. Fig. 2.1 shows the UP core workflows, depicting their relative weight when iterating through the

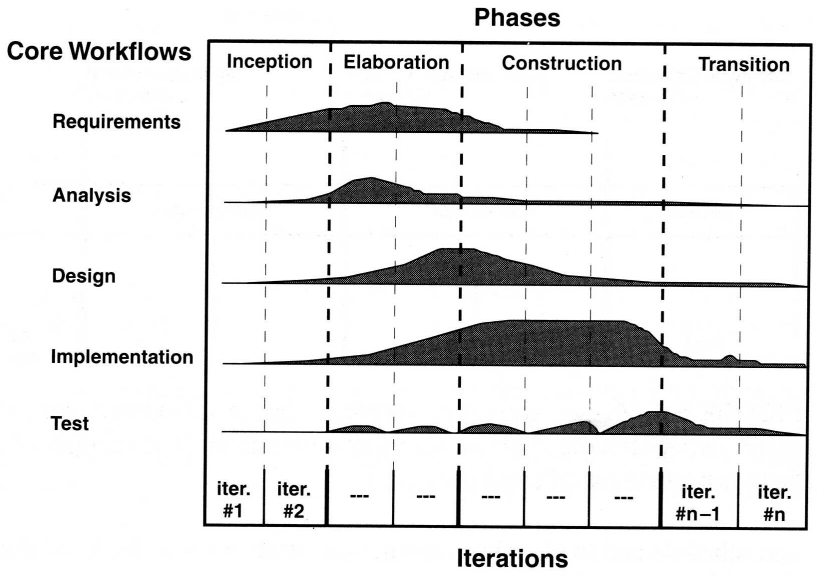

Figure 2.1: Unified Process core workflows and their relative weight through the iterations from requirements capture until testing (taken from [JBR99]).

defined phases.

A software engineering method is a structured approach to software development that facilitates the production of quality software in a cost-effective way [Som07]. Examples of software engineering methods are Yourdon/Demarco's Structured Analysis [You88, Dem79], the Booch Method [Boo93], Jacobson's OOSE [JCJv92], or Rumbaugh's OMT [RBL+90], amongst many others. These last three, together with other contributions, have been integrated into a unified approach called the Unified Modeling Language (UML) [RJB99, SV05].

All methods are based on the development of system models, composed of several artifacts (sub-models), each one giving a different point of view of the system model.

2.2 Software Modeling

A software model is an abstraction of a software system that may already exist or is to be built. A software model is at a higher abstraction level than the software system itself, and so it is easier for human software designers and/or developers to reason about the software system by using a software model.

In the requirements engineering phase of a software development process, the user requirements that are about the system's functionality are usually modeled in a form that may be more or less formal and rigorous. Typical types of models used nowadays are visual models (based on diagrams) [RJB99] and formal models

(based on formal methods and notations with a mathematical foundation) [Far07, FL98, Mey06, SWM06]. Then, traditionally, by manual or computer assisted model refinements, software engineers produce an analysis model, that maps the problem domain, and then a design model, that models a software solution for the previously modeled problem. Based on the design model, developers may then code and test the designed solution [Pre05].

2.3 Design by Contract

Design by Contract (DbC), aims at developing software that is trustable, in which people can rely on. In DbC, a contract is a means by which two elements agree in an explicit roster of clauses for mutual obligation and benefits [SWM06, Mey06, Art04b, Art04a].

In an object-oriented setting, components and classes act as providers and clients of services. Through a contract, two components or classes agree in a protocol for communicating, that is, which methods are exported (publicly visible) by the intervening classes and what type are its parameters and results. A contract not only specifies the protocol for instances of classes or components' communication, the syntax of the communication, but also specifies what conditions have to be met in order for the communication to happen, that is, what conditions must the caller (client) assure in order to be allowable to call a given method (preconditions) and what conditions must the callee (provider) guarantee that hold after the method is returned (post-conditions), i.e. the semantics of the communication.

A contract may also specify conditions that must always hold, as invariant conditions that are imposed to the state of a class' instances. DbC can be viewed as a means to develop a system's platform independent model and, as it is abstractly but formally defined, the room for ambiguity is strongly reduced, meaning that it may be machine understandable and that it may be used for model transformation activities. This way, DbC can be an approach to model driven development.

The way of specifying contracts in UML based class models is by using OCL (Object Constraint Language). OCL enables, amongst other things, the specification of invariant conditions in the context of a class, and the specification of preconditions and post-conditions in the context of a class' operation.

2.4 Model-driven software development

2.4.1 MDD basics

Model-driven software development (MDSD), or just Model-driven development (MDD), is a paradigm for developing software systems in which models are first

class citizens. In fact, model-driven software development is, like its name suggests, driven by the activity of modeling a software system [WBP+03], and so code is renegaded to a second plan.

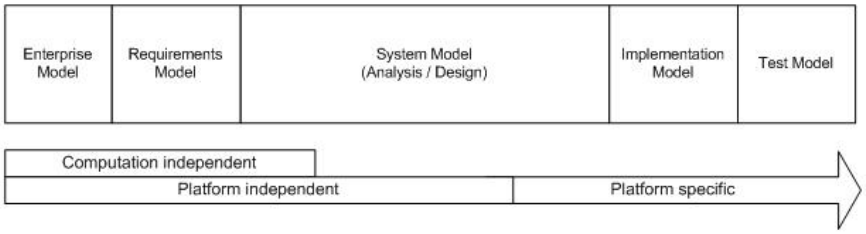

Figure 2.2: From more abstract to more concrete models.

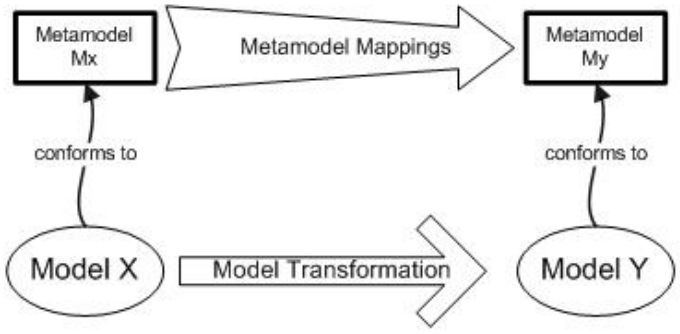

Figure 2.3: Model transformation conforms to defined mappings between source and target metamodels.

MDD approaches software development by constructing models that may be refined (transformed) through different levels of abstraction, from a platform independent level, or a computation independent level, to a platform specific model that is directly mapped to final code (see figure 2.2).

The first models constructed in a MDD process are platform independent models (PIM), meaning that they model a system in a platform independent way. A PIM can be defined in a computation independent way, meaning that besides being platform independent it is also computation independent. A computation independent model (CIM) does not model how to make, or compute, things, but only what is expected to be made or computed, whilst a PIM may prescribe how to make computations provided it is made in a platform independent way. A CIM is typically only dependent of domain or business concerns. This way, a CIM is independent of implementation platforms and describes the problem from

the point of view of the business or domain environment. It is a model of the business to where a software system is going to be built [SV08].

A model may be composed of several model artifacts, each one modeling a different view of the system. Platform independent models may be the only models made by the modeler's hand. Indeed, MDD prescribes the definition of model-to-model (M2M) transformation processes that enable the transformation of PIMs to other PIMs or to platform specific models (PSMs). When a CIM exists, a default PIM may also be generated by a M2M process.

A PIM may model a software system by modeling its structural, functional, behavioral and presentation aspects. It consists of a system specification in a platform independent way. At any moment, a given aspect or view of the system may be obtained from the others by a M2M transformation process that implements a set of mappings that are defined between the respective metamodels (see figure 2.3). When the PIM is satisfactory, the next step is to transform it into a platform specific model (PSM), which is a model of the system making use of a given platform specific issues or features. From a PSM, code may be automatically generated by using a model-to-code (M2C) transformation process, that is a code generation process.

At any moment, a model may be subject to model-to-text (M2T) generation for documentation purposes. Acronyms M2C and M2T are typically used interchangeably, as code can be understood as a special case of text.

MDD enables the collection of the knowledge acquired by a company through its entire life, and its storage in the form of models. These models may, then, be adapted and transformed as new business environments are taking place or technological changes occur [Ins06, PB08].

M2M and M2C processes operate over models, but must be defined as functions from one model type to another model type. A model type is defined by a metamodel, which is a model that defines a model. A given model must always conform to a defined metamodel. Figure 2.4 presents the metamodel levels in OMG's MDA. At the top level (M3), there is the Meta-Object Facility (MOF), which is a standard language for defining metamodels. Below the meta-metamodel level, where MOF is defined, other MOF-based metamodels may be defined. This is the meta-model level (M2), and is where the Unified Modeling Language (UML), and other metamodel-level languages, are defined. The MOF conforms to itself, and it is defined according to its own definition. UML is defined through MOF, and it conforms to the MOF. The model level (M1) is where models are defined by using a metamodel defined in the above level. Each UML model, made for some specific domain, conforms to UML, and is at metamodel level M1. A UML model defines a domain specific language (DSL) for modeling instances of that domain. A specific model instance is at metamodel level M0.

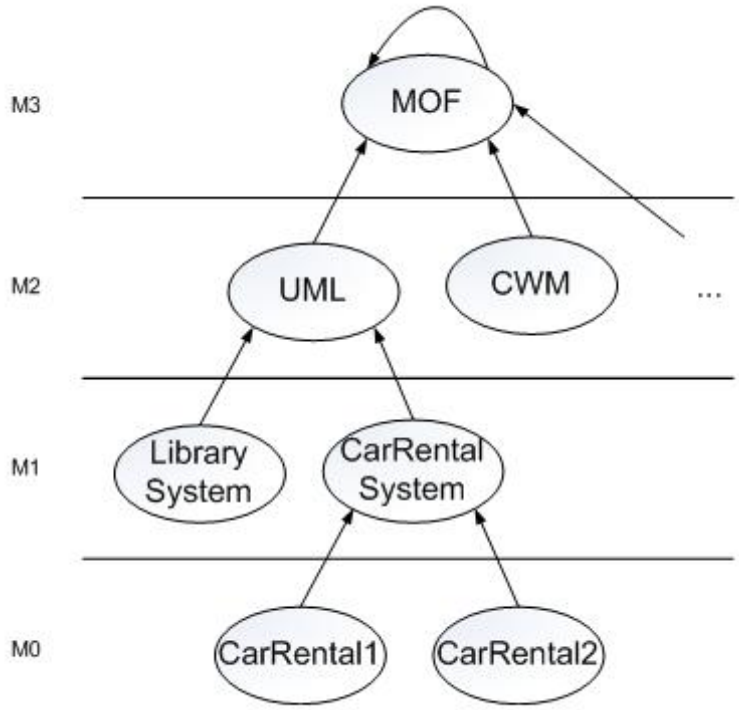

Figure 2.4: Metamodel levels defined for OMG's MDA.

2.4.2 Modeling for MDD

When modeling for MDD, the software modeler must take into account that models are not only for human reasoning about the system being built, but are mainly for enabling the automation of some software process steps or activities. For that purpose, models must be rigorous, complete and unambiguous, and executable or, at least, translatable to a given target programming language.

Let's illustrate this issue with an example. Figure 2.5 shows the domain classes diagram developed for managing books, borrowers and loans in a library. A registered library user (*LibraryUser*) is either a librarian or a borrower. A *Book* has an ISBN code, a title and an author and there may exist several book copies (*BookCopy*) of a given book, in the Library. A *BookCopy* has an identifying book copy code and a status, from a *BookCopyStatus* enumerable. *Borrower*s may make *Loan*s of book copies. A given business constraint is that a library user may not define his/her *Password* equal to his/her *Login*. The UML class diagram, by itself alone, is incomplete in this point, giving room to ambiguity.

The model needs to be enriched with comments and/or constraints. A common way to do this is to use UML comment symbols with natural language, as in figure 2.6. But, for being complete and translatable, in order to be automatically

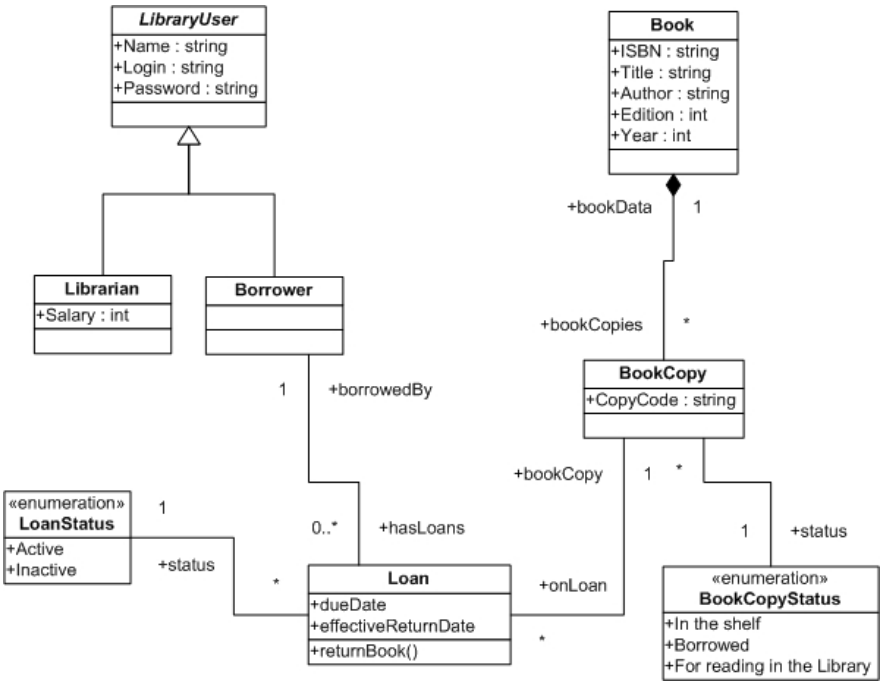

Figure 2.5: UML class diagram for a LibrarySystem example.

processed, the system's domain class model must have the UML class diagram complemented with formal and unambiguous constraints.

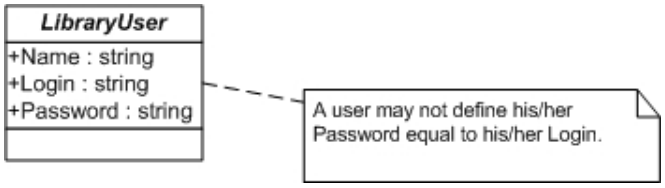

Figure 2.6: Comment (informal constraint) in UML.

For that purpose, it is needed a formal language, such as the object-constraint language (OCL), to rigorously specify the constraints. The business constraint in figure 2.6 could, for instance, be specified in OCL like:

```
context LibraryUser
  inv:  self.Login <> self.Password
```

OCL is part of the UML standard, and allows the definition of constraints over object models, enabling the use of design by contract (see section 2.3) within a UML setting. Using OCL it is possible to:

- Define constraints over the values of the attributes of a given class' instances. The conjunction of all the constraints of a class is known as that class' invariant.

- Define the conditions that have to hold before invoking a class' operation, that is the precondition for that operation.

- Define the conditions that must hold after returning from a class' operation, that is the post-condition for that operation.

As seen before, a class definition, involving the specification of its attributes and respective types, its operations and respective signatures, preconditions and post-conditions, and the class' invariant, is known as a class contract.

Amongst other things, OCL also makes possible to define derived attributes and initial attribute values (default values), mutual exclusion conditions and subset relations over associations, and define guard conditions over state transitions, in state-machine diagrams.

For defining computations and actions within UML models, an action language may be used. OMG defines a standard abstract action language for UML models, called Action Semantics. Actions Semantics is further discussed in section 2.5.1.

2.4.3 Model-transformation

The aforementioned model transformation processes, namely model-to-model (M2M) and model-to-text (M2T), have different purposes [Jéz05]. Model-to-text transformations aim the generation of code or documentation (e.g.: Java, XML, HTML), and should be limited to syntactic level transformations (e.g.: from a PSM to code, or from a PIM to documentation in HTML). Model-to-model transformations aim the refinement of models, its refactoring, reverse engineering, application of patterns, generation of new views (PIM-to-PIM), PIM-to-PSM transformations, or any model engineering activity that can be automated.

M2T techniques can be divided in [Jéz05]:

- Visitor-based approaches: traverse the internal representation of a model (abstract syntax tree) and write code to a text stream.

- Template-based approaches: are based on the construction of templates, which consist of the target text containing slices of meta-code to access information from the source model.

M2M techniques comprise [Jéz05]:

- General purpose programming languages (e.g.: Java, C♯)

- Generic transformation tools (e.g.: XSLT)
- CASE tools scripting languages (e.g.: Rose)
- Dedicated Model transformation tools (e.g.: ATL & MTL (INRIA), QVTEclipse)
- Meta-modeling tools (e.g.: MetaCase, Kermeta)

Whatever the M2M technique, it may provide a declarative or an imperative paradigm and language for specifying model transformations. Declarative languages describe relationships between variables in terms of functions or inference rules, which are then inputted to an algorithm with an inference engine to produce a result. They specify what to do, or what shall be the relations between source and target models. Imperative languages specify explicit manipulation of the model instances, stating how to do the transformation, or how to derive a target model from the source one [Jéz05].

Declarative and imperative paradigms may be combined into an hybrid style, that uses a declarative precondition, used to identify elements that may trigger an imperative transformation rule, and finally apply a declarative post-condition to the model elements generated by the application of the rule [Jéz05].

M2M Execution Strategy

Model-transformation execution strategy is based on the invocation of transformation rules, which are units responsible for transforming a particular selection of the source model to target model elements.

Rules may be defined through declarations, which are binary relations, with a left-hand-side (LHS) and a right-hand-side (RHS), that relate elements in the LHS with elements in the RHS models. Another way of defining rules is through implementations, which are imperative specifications of how to create target model elements from source model elements [Jéz05].

When applying the transformation rules, the execution engine, that depends on the selected model-transformation technique, searches the entire source model abstract syntax tree (or metamodel instance) looking for a match.

A pattern match occurs when elements from the LHS model are identified as meeting the constraints (pattern) defined by a transformation rule. When this is the case, a match triggers the creation or update of model elements in the target model.

A transformation rule, triggered by a match, may be invoked explicitly or implicitly. In the first case, the transformation rule is called via the invocation of operations (Java like). In the latter case, the invocation of the transformation is done implicitly, based on the context and rule's signature [Jéz05].

The execution strategy of the transformation rules is typically structure-driven, meaning that the first applicable rules will create the hierarchical structure of the target model, and only then rules for setting attributes and references in the target model are applied. Other execution strategies will probably combine different techniques, and so are called hybrid approaches [Jéz05].

2.4.4 Advantages of MDD approaches

Model-driven development approaches try to solve the following problems [WBP+03, SV08]:

- The productivity problem – Today's software projects productive activities are coding and testing. For that fact, software team members spend little time modeling or documenting systems. This becomes a problem when the team is dismantled and other people needs to maintain the software (e.g. fix bugs, enhance functionality). MDD tries to solve this problem by transfering the productive activities from coding and testing to modeling, and basing the software development process in model-to-model and model-to-text automatic transformations. Models, being at a high abstraction level, are easier for newcomers to understand, lowering the effort and cost of software maintenance.

- The portability problem – Every year new technologies become popular (e.g.: Java, C#, .Net, XML, SOAP, etc.) and, either because they solve real existing problems, or because tool vendors stop supporting old technologies, or even just because customers want to, systems often need to be ported to new technologies. Consequently, previous investments in the older technologies lose value. MDD, helps maintaining the value of previous investments by basing the production of code in automatic model-to-code transformation processes.

- The interoperability problem – In an organization there are, typically, several software systems that need to interoperate. MDD leverages the notion of component from code to models, i.e. models may be component oriented, and code generated from models must be interoperable. Other software systems also are generated from their models, so interoperability at model-level is maintained at code-level.

- The maintenance and documentation problem – Writing documentation has always been considered by software developers as a waste of time, because they feel their main task is to produce code. Also, when projects get late, what is most likely to be cut off from the project's deliverables is documentation. The lack of documentation, or its low quality, has the consequence of complicating the software maintenance, making it more costly and taking

more time to accomplish. Having documentation and code desynchronized also promotes the outdating of documentation when maintaining the code.

Solutions to this problem have involved the generation of documentation from code, but this only solves the problem for low-level documentation. Higher level documentation and models are rarely updated during maintenance. MDD tries to solve this by having code produced automatically from models. This models can, for instance, be embedded into high-level documentation in a *literate programming* fashion [Knu92], like enabled by VdmTools for VDM specifications [FL98].

2.5 UML Metamodel

To help understand the metamodel presented in this dissertation (in chapter 4), a brief description of relevant elements of the UML metamodel is made in this section.

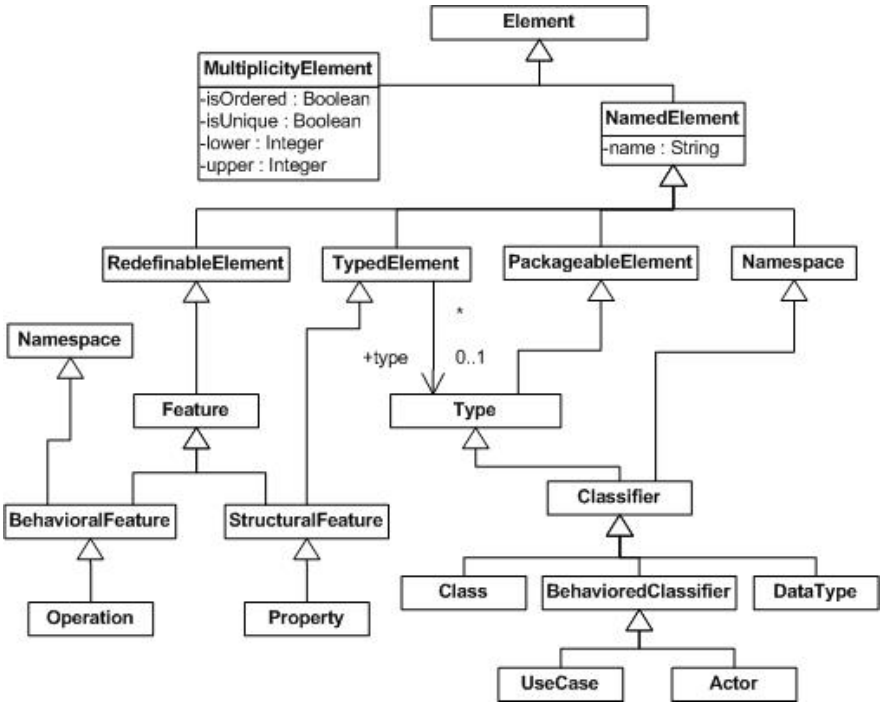

Figure 2.7: UML Metamodel top-level elements

Figures 2.7 through 2.11 show UML metamodel elements from the UML Infrastructure Library, or Core package [OMG09a] and from the UML superstructure [OMG09b].

The top UML hierarchy is *Element*. An element may have multiplicity, order or uniqueness characteristics, in which case it is a $MultiplicityElement$. A $NamedElement$ is an *Element* with a name.

A $TypedElement$ is a kind of $NamedElement$ that represents elements with types. A $TypedElement$ may optionally have no type at all. *Type* is a *PackageableElement*, which in turn is a named element that can be owned by a package (not represented in the figure).

A $Namespace$ is a named element that can own other named elements. Each named element may be owned by at most one namespace. Named elements can be identified by name in a namespace either by being directly owned by the namespace or by being introduced into the namespace by other means (e.g., importing or inheriting) [OMG09a].

A $Classifier$ is a namespace whose members can include features. And, a $Feature$ is a redefinable element that declares a behavioral or structural characteristic of instances of classifiers. A $RedefinableElement$ is a named element that can be redefined in the context of a generalization.

A *Class* is a classifier whose features are attributes and operations. Attributes of a class are represented by instances of *Property* that are owned by the class. Some of these attributes may represent the ends of associations (see figure 2.10). *Property* is a structural feature of a classifier. An *Operation* is a behavioral feature of a classifier.

A $BehavioredClassifier$ (from the UMLsuperstructure) may have an interface realization. From this point of view, a *Class* is a behaviored classifier, and it is represented as so in the UMLsuperstructure (although not in the UML infrastructure).

Actor and $UseCase$ are also behaviored classifiers.

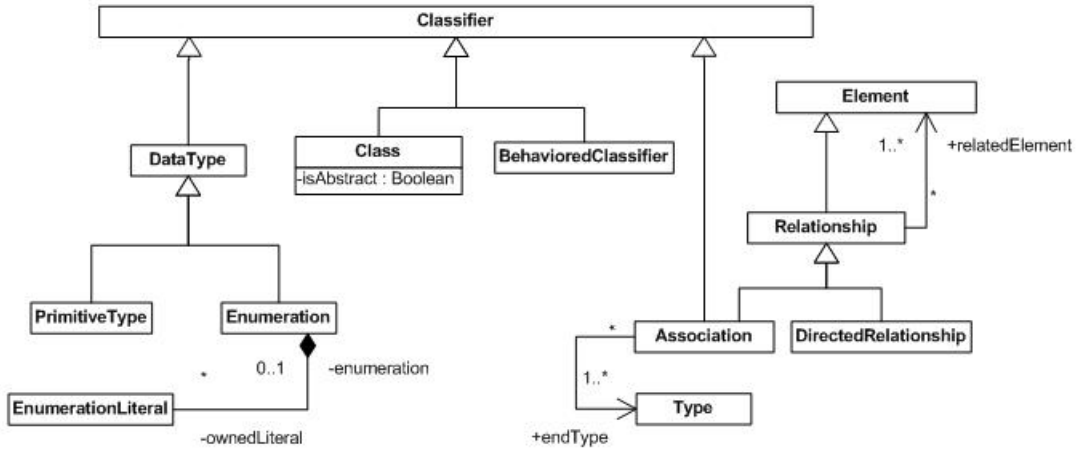

Figure 2.8: UML Metamodel - Classifiers and Relationships

Classifiers are types. A *DataType* is also a classifier (see figure 2.8). A data type may be a primitive type, which may be an Integer, Boolean, String or UnlimitedNatural (not represented in figures), or an Enumeration.

A *Relationship* is an element that relates other elements. In particular, a *DirectedRelationship* relates a source and a target *Element*.

An *Association* is a relationship and a classifier, which relates typed instances. It has at least two ends represented by properties, each of which is connected to the type of the end. More than one end of an association may have the same type [OMG09a].

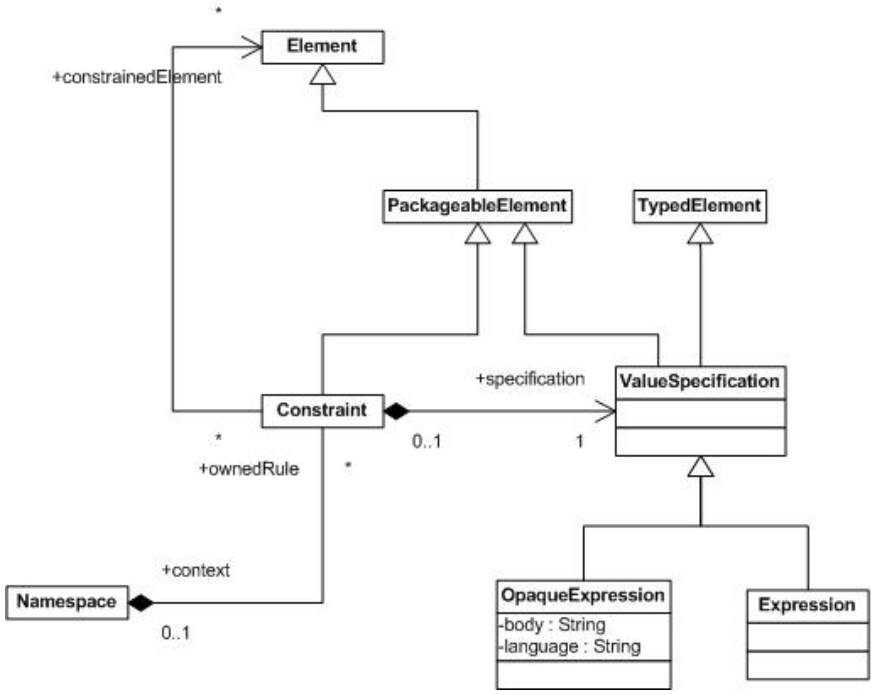

Figure 2.9: UML Metamodel - Constraints

A *Constraint* is a *PackageableElement* defining a rule that constrains one or more *Element*s (see figure 2.9). The rule is specified through a boolean value specification, by means of an *OpaqueExpression* or an *Expression*.

From figure 2.10, a *Class* is comprised of properties, operations and constraints. An operation may also be constrained through its preconditions or post-conditions. An operation may have parameters and a result type.

In figure 2.11, one can see that a *UseCase* is defined over a subject, composed of several UML elements. A use case may have extension points where *Extend*s from extension use cases can be plugged. An *Extend*, which is the metaclass of a use case extension relation, may have a (pre)condition.

Also, a use case model may have include relations (*Include* instances) that add use cases to an including use case.

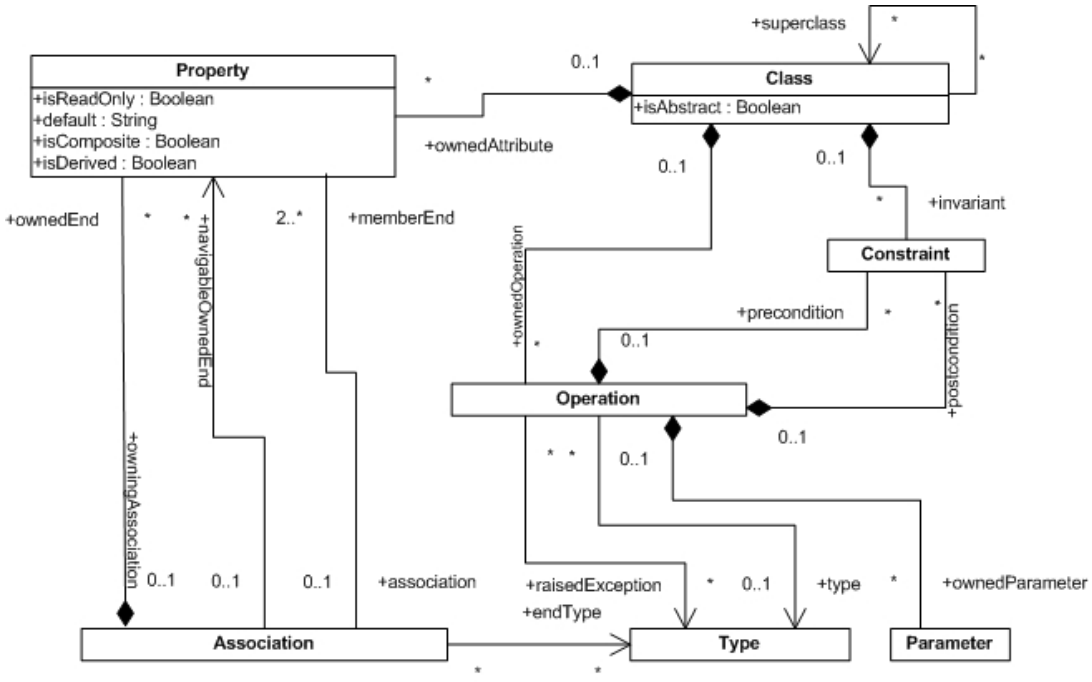

Figure 2.10: UML Metamodel - Class model elements

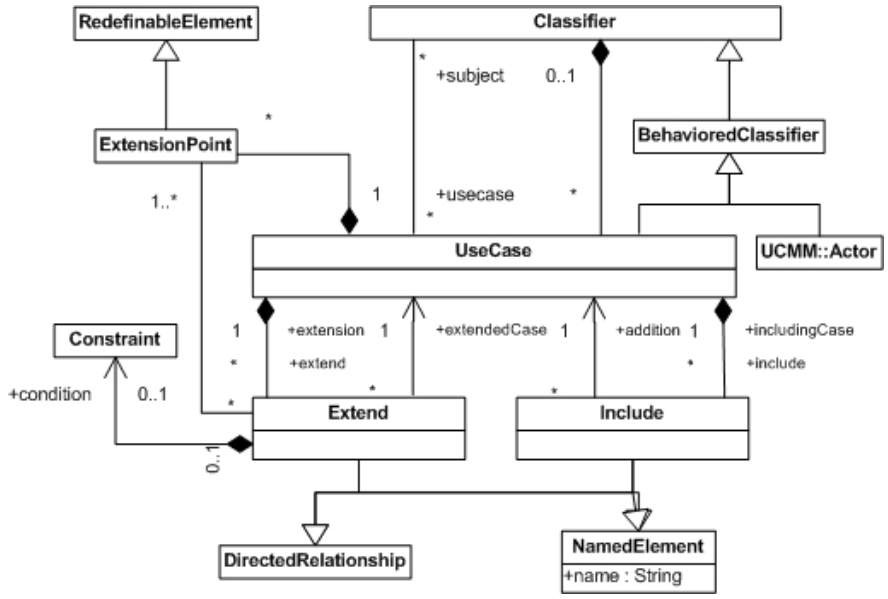

Figure 2.11: UML Metamodel - Use Case model elements

A complete description of the UML metamodel may be found by combining [OMG09a] and [OMG09b].

2.5.1 Object Constraint Language and Action Semantics

Besides the Object Constraint Language [OMG03], referred in sections 2.3 and 2.4.2, which allows the definition of constraints over UML models, the OMG also defined the abstract syntax for an action language (referred to as Action Semantics in [OMG09b] chapter 11 and [OMG01]).

Action Semantics (AS) defines an action language for modeling computation within UML models. In fact, only the abstract syntax of AS is defined by the OMG, so several different concrete languages (or surface action languages) could be specified in a manner that all could concretely represent the same AS abstract syntax tree.

An action is the fundamental unit of behavior specification [OMG09b]. It takes a set of inputs and converts them into a set of outputs, though both sets may be empty.

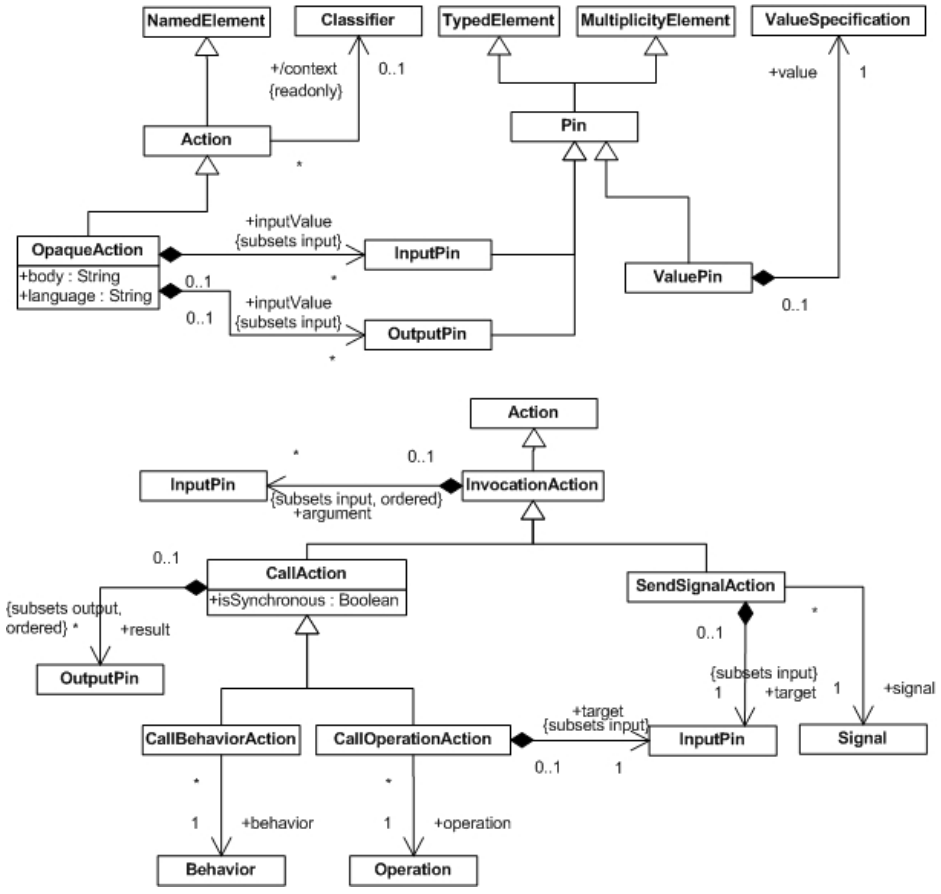

Figure 2.12: UML Metamodel - Basic actions

An action may modify the state of the system in which it executes. The values that are the inputs to an action may be described by value specifications, obtained from the output of actions that have one output (see Fig. 2.12). The

activity flow model supports providing inputs to actions from the outputs of other actions.

Actions are contained in behaviors, which provide their context and may constrain actions to determine when they execute and what inputs they have.

Basic actions include operation calls, signal sends, and direct behavior invocations (CallOperationAction, SendSignalAction and CallBehaviorAction in Fig. 2.12). An operation may be bound to activities, state machine transitions, or other behaviors, which specify the way the operation modifies its context state

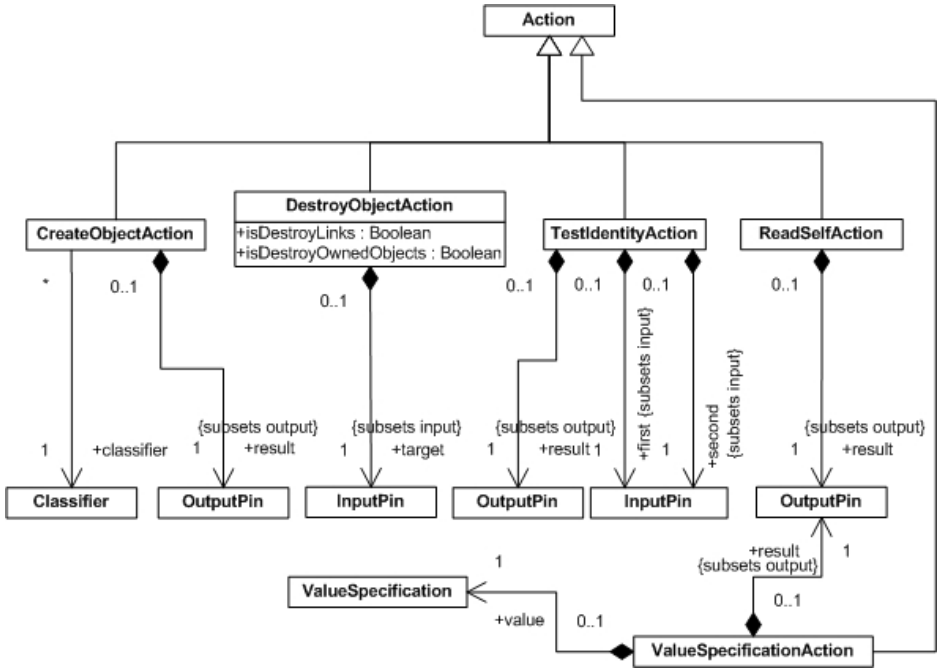

Figure 2.13: UML Metamodel - Object actions

The intermediate level describes the various primitive actions, which may be divided into the ones that carry out a computation, and the ones that access object memory. There is no intermediate level action that does those two things simultaneously.

Fig. 2.13 shows some of the classes for Actions of the intermediate level that enable object manipulation (object actions) as, for instance, creating and destroying objects. Object actions create and destroy objects, while structural feature actions support the reading and writing of structural features. Objects, structural features, links, and variables have values that can be read and written by structural feature actions.

A concrete action language, or a surface action language, as the UML specification refers to it, could comprise both primitive actions and the constraints provided by behaviors, and additionally it could map higher-level constructs to the actions (e.g.: it could define a creation operation with initialization that

would map to the primitive create action to create the object, and further actions to initialize attribute values and create objects for mandatory associations). This way, modelers can work in terms of higher-level constructs as provided by their chosen surface language or notation [OMG09b].

Other Actions are defined for accepting events, including operation calls, and retrieving the property values of an object all at once. The StartClassifierBehaviorAction provides a way to indicate when the classifier behavior of a newly created object should begin to execute.

Action Semantics is platform independent and doesn't enforce constraints, initialization or operation propagation, during execution. To test constraints is controversial, because not all constraints are valid at all times.

As partially seen before, AS is able to access attribute and link values, the self value, and is able to manipulate collections [OMG09b].

When an action violates aspects of static UML modeling that constrain runtime behavior, the semantics is left undefined (e.g.: attempting to create an instance of an abstract class), with the exception of the lower multiplicity bound, which is ignored in the execution of actions and no error or undefined semantics is implied. Otherwise, it is impossible to use actions to pass through the intermediate configurations necessary to construct object configurations that satisfy multiplicity constraints.

The UML specification [OMG09b] treats all actions as executing concurrently unless explicitly sequenced by a flow of data or control. In addition, each action is defined so that it is free of any context that describes how it is used, so that data access and other computations are two separate primitive actions connected together when required, and each action is unaware of the source or destination of the data.

2.6 Package dependency "merge" relation

The UML2 specification [OMG09a, OMG09b] introduces a new package relation, package merge, which enables the separation of concerns into different packages.

A package merge is a directed relationship between two packages indicating that the contents of the two packages are to be combined [OMG09b] in a way that the source element conceptually adds the characteristics of the target element to its own characteristics, resulting in an element that combines the characteristics of both.

The package merge mechanism is used when elements defined in different packages have the same name and are intended to represent the same concept. It enables different definitions of a given concept to be given for different purposes, starting from a common base definition. A given base concept is extended in increments, with each increment defined in a separate merged package [OMG09b]. It is possible to obtain a custom definition of a concept for a specific end, by

selecting which increments to merge. Package merge is extensively used in the definition of the UML2 metamodel.

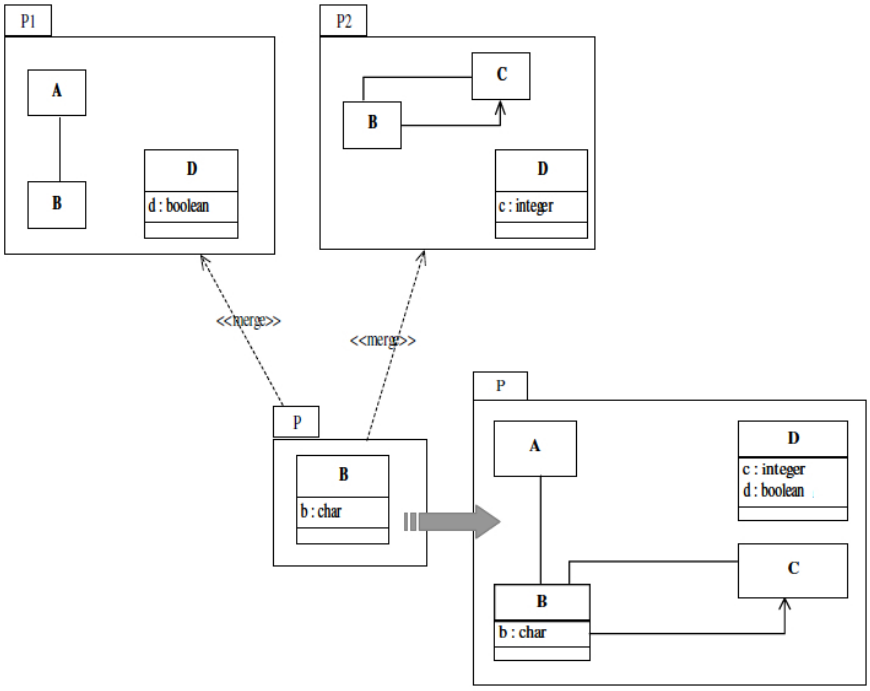

Figure 2.14: Package merge between packages P1, P2 and P (taken from [AD06]).

Fig. 2.14 illustrates the package merge between P1, P2 and P. The package merge relation can also be viewed as an operation between packages. It takes two packages as parameters and produces a new package that combines the contents of the parameter packages involved. There is no difference between the semantics of a model with explicit package merges, and a model in which all the merges have been performed [OMG09b].

2.7 Task Analysis and Modeling

Task analysis is a technique used within the HCI community to analyze the way people act when performing their jobs. Task analysis can be approached by the following ways [DFAB98]:

Task decomposition Decomposes tasks into subtasks, taking care with the order these are performed (eg.: HTA - Hierarchical Task Analysis). See figure 2.15.

Knowledge-based techniques Focus on what the users need to know about the objects and actions involved in a task, and how

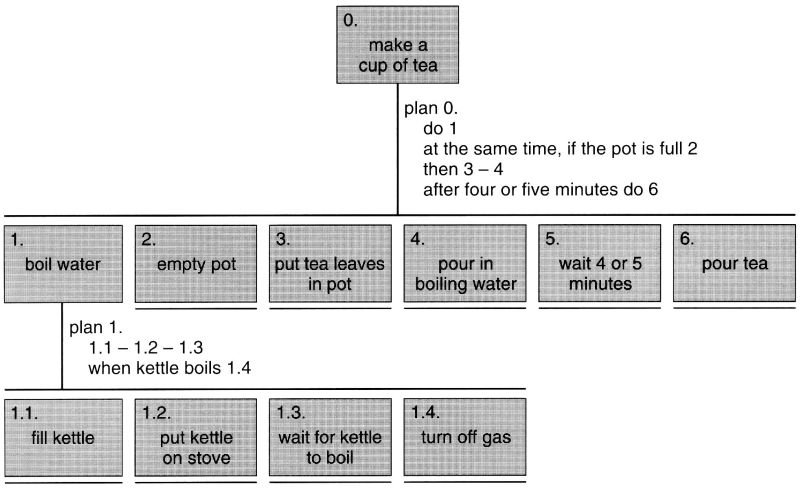

Figure 2.15: Task hierarchy for making a cup of tea(taken from [DFAB98]).

that knowledge is organized (eg.: TAKD - Task Analysis for Knowledge Description).

Entity-relation-based analysis Puts an emphasis on identifying actors and objects, the relationships between them and the actions they perform (eg.: ATOM - Analysis for Task Object Modeling Method).

2.7.1 ConcurTaskTrees

ConcurTaskTrees is a graphic notation for specifying task models. It has a hierarchical structure, just like hierarchical task analysis, which enables reusable task structures to be defined at both a low and a high semantic level. ConcurTaskTrees enables the use of operators that link subtasks at the same abstraction level, which describe a temporal relationship between tasks. This sort of aspect was not usually formally present in task models. By allowing the modeler to use these operators it is possible to express clearly the logical temporal relationships, which shall be taken into account in the user interface implementation to allow the user to perform at any time the tasks that should be active from a semantic point of view.

The operators used by CTT to describe the temporal relationships are [PMM97, Pat03]:

- T1 ||| T2, interleaving: the actions of tasks T1 and T2 can be performed in any order;

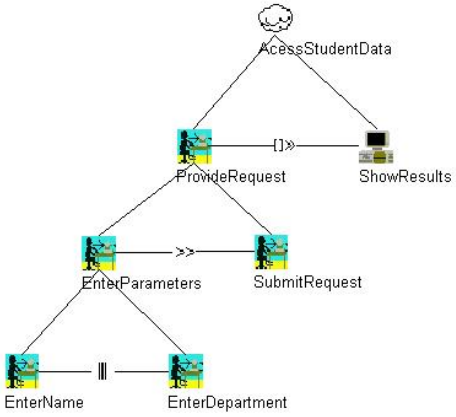

Figure 2.16: Example of a CTT task model (taken from http://giove.isti.cnr.it/tools/ctte/).

- T1 |[]| T2, synchronization: tasks T1 and T2 must synchronize on some actions in order to exchange information;
- T1 >> T2, enabling: when task T1 is terminated, task T2 is activated;
- T1 []>> T2, enabling with information passing: besides activating T2, the termination of T1 yelds some value to T2;
- T1 [> T2, deactivation: when one action from task T2 occurs, T1 is deactivated;
- T1*, iteration: the task is iterative;
- T1(n), finite iteration: the number of times the task is performed is specified;
- [T1], optional task: task execution is not mandatory;
- T1 recursion: task T1 may include itself in the task specification.

An example of a task model using the CTT notation is shown in figure 2.16.

2.8 User Interface Development

There are several user interface (UI) development methodologies and tools, like:

- WYSIWYG (What You See Is What You Get) oriented UI builders and IDEs (e.g. Microsoft Visual Studio, Eclipse, NetBeans, NextStep);

- markup languages (e.g. HTML and XML-based UI description languages, such as UIML, XUL and XAML); or,

- model-based user interface development environments - MB-UIDE - (e.g. HUMANOID, UIDE, MECANO, Teallach, etc.) [Pin00, Pin02, Pai06].

Interaction between the user and the system takes place on the UI. The interface style affects the nature of this dialogue. Common interface styles include [DFAB98]:

1. Command line interfaces (CLI) - Provide a way to express direct instructions to the computer, or an application. CLIs can be very powerful and flexible for experienced users, but are usually difficult to learn and use by non-experienced users.

2. Natural language - This kind of UI style can, for now, be used only in restricted domains, and with a restricted number of allowed phrases, because of the natural ambiguity of natural languages.

3. Menu-driven interfaces - Offer a set of options available to the user, that are selected using the mouse, or numeric or alphabetic keys. Menus are hierarchically ordered. This kind of systems can be purely text based or can be a restricted form of a full WIMP interface style.

4. Question/answer and query dialog - This interface styles offer a simple mechanism for providing input to a domain specific application. The user is asked a series of questions, being the dialog driven by the answer to the previous questions.

5. Form-fills and spreadsheets - Form-filling UI style is primarily used for data-entry applications. Business applications, typically use this kind of interface, because it resembles traditional paper forms, and are usually easy to learn and use. Spreadsheets are a variation of a form filling system, where a grid of cells is provided for entering values and formulae, and the system is responsible for maintaining the consistency of the values displayed.

6. Windows, Icons, Menus, and Pointers (WIMP) - This UI style is also simply known by windowing systems or window-based systems. This is currently the most common interface style for interactive applications.

7. Point-and-click - This interface style overlaps some of WIMP aspects, but it is simpler and is not tied to mouse-based interfaces, being extensively used in touchscreen applications. It is often combined with a menu-driven interface.

8. 3D interfaces - This style spans from simple WIMP applications with 3D appearance elements to complex 3D workspaces or virtual reality systems.

A given UI can combine one or more of the referred interface styles. What is commonly called a Graphical User Interface (GUI) is an UI that mixes the WIMP interface style with any other.

UI is also affected by dialog design and layout. The way information is presented to the user and the screen layout for entering information have important effects on system usability.

User Interface Models (UIM) can be used to model a user interface of a given system. *Dix et al.* identify the following set of UI model concerns [DFAB98]:

- The definition of allowed system UI states and transitions
- The specification of allowed sequences of user events and system events on the UI
- The establishment of a link between the events on the UI and the core system's functionality
- How is the information (abstractly) presented to the user
- What is the concrete aspect of that presentation

The concerns identified above are addressed by disparate UIM composing models, or sub-models or model views.

Martikainen [Mar02] defines a user interface model (UIM) as "a declarative specification of a user interface (UI), including its appearance, the connections between its elements or how it interacts with the underlying application functionality". A UI model represents all the relevant aspects of a user interface in some type of interface modeling language or notation. UI models are generally task-oriented and use high abstraction levels to achieve device independence and UI description reuse [PE99, Mar02].

Several approaches for systematizing the gap bridging between user tasks, or usage scenarios, and the application (business-logic) model are addressed by [CWNL03, Con95, Con06a, EKK06, KLM03, MAT01].

Just like in any interactive system, in Window-based systems (Web-based included), one can distinguish three levels of specification detail [DFAB98]:

Lexical level - The allowed "words" (interaction objects) for the interaction.

Syntactic level - The layout of abstract interaction objects (AIOs) and the order of events allowed.

Semantic level - The meaning of each event or interface state transition and how it maps to the core system's functionality.

Ergonomics (or Human-Factors) in HCI are concerned with usability issues in UI modeling. Usability issues are concentrated on the lexical and partialy the syntactic level of the UI model. Typically, those issues are addressed by the UI concrete presentation model and the task model. Those models are components of the whole UI model. For that reason, models that compose the whole UI model are sometimes called 'component models', submodels or model views.

Dix et al. identifies three principles of usability that may be applied during the design of a system's UI [DFAB98]:

Learnability - "the ease with which new users can begin effective interaction and achieve maximal performance."

Flexibility - "the multiplicity of ways the user and system exchange information."

Robustness - "The level of support provided to the user in determining successful achievement and assessment of goals."

2.8.1 Canonical Abstract Prototypes

Canonical Abstract Prototypes (CAP) are an approach and notation, proposed by L. Constantine [CL99, Con03], for capturing the presentation aspects of interactive systems. Canonical Abstract Prototypes capture only the abstract presentation aspects of a user interface, by making use of abstract interaction objects (AIO), which are UI elements that don't have a unique concrete representation.

CAP are based on 3 extensible generic universal symbols [CN04, Con03]:

1. Material (or generic container): represents information, data or other objects shown to the user during a task.

2. Tool (or generic action/operation): represents UI objects that can be used to manipulate, control or transform materials.

3. Hybrid (or active material): represents UI components with characteristics both from materials and tools like, for instance, editable fields or lists of selectable items.

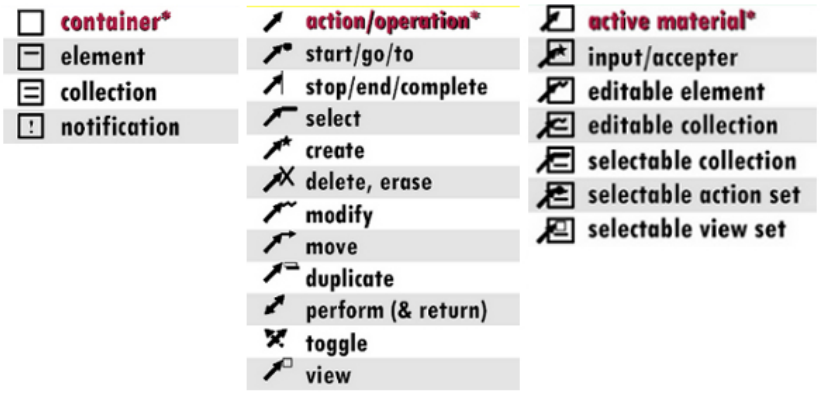

Figure 2.17: The basic symbols and its extensions, for Canonical Abstract Prototypes (adapted from [Con03]).

Fig. 2.17 shows the main symbols of the canonical abstract notation, that allow the development of UI abstract presentation models, like the one shown if Fig. 2.18. The figure shows a prototype of a browsable selectable list of messages and the output of detailed information about the selected list item. The symbol >>> represents repetition and, in the example, it means that the aligned elements in the Inbox selectable collection are repeated in every line.

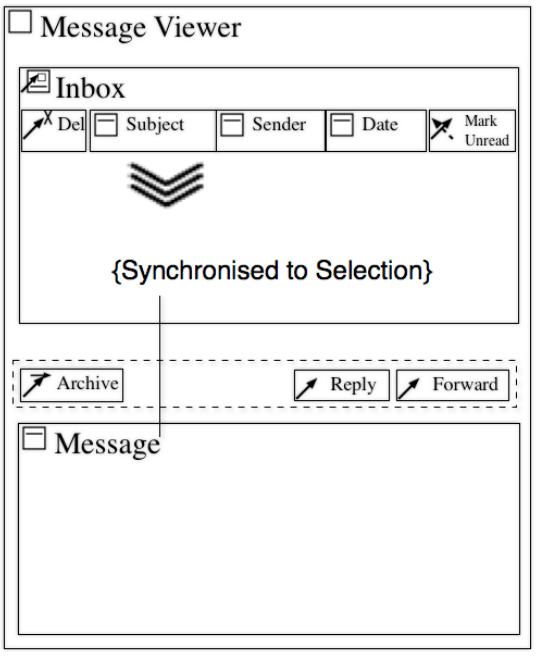

Figure 2.18: Example of a CAP for a Message viewer (taken from http://www.mcs.vuw.ac.nz/courses/comp311/2008t2/comp311cap2008.pdf).

2.8.2 XML-based User Interface Description Languages

Some XML-based user interface descriptions provide an abstraction level that is implementation independent, until a concrete presentation is needed. As XML-based approaches for UI description, one can consider, amongst others [Fer05, APB+99]:

- XIML (eXtensible Interface Markup Language);
- XUL (XML-based User-interface Language);
- UIML (User Interface Markup Language), and
- XAML (eXtensible Application Markup Language).

XML-based declarative UI description languages also use several components, each capturing a different view from the UI or the application presentation layer. Typicall components in XML-based languages are:

Content and Layout - Description of the contents of the user interface, without referencing any concrete presentation object, that is an abstract presentation model.

Concrete Presentation - Relation from abstract content elements to concrete presentation objects.

XIML - eXtensible Interface Markup Language

XIML aims at providing "a standard mechanism for applications and tools to interchange interaction data and to inter-operate with integrated user-interface engineering processes, from design, to operation, to evaluation..."[1]. This XML-based UI language allows for the description of a graphical user interface in several interface components. There are five basic interface components, three of which are contextual and abstract [Fer05]:

task - captures a given user task suported by the GUI.

domain - captures the application domain into a hierarchy of data objects and classes.

user - captures the user profiles into a hierarchy.

The other two are implementation specific [Fer05]:

dialog - defines the allowed interaction actions and the possible navigational flow between those actions.

presentation - defines a set of concrete interaction objects (*widgets*) to be related to the abstract domain components.

[1] XIML Forum: http://www.ximl.org

XUL - XML-based User-interface Language

XUL (pronounced *zool*) has been developed for the *Mozilla* project[2] and is directly supported by the Firefox browser. A XUL UI description file also allows a separation between the abstract structure of the interface and its concrete presentation elements [Fer05].

UIML - User Interface Markup Language

UIML has been proposed by Harmonia[3] and is being standardized by OASIS[4] (Organization for the Advancement of Structured Information Standards). This XML-based UI description language follows the principle "one application, multiple interfaces", and aims at supporting the development of "device-independent and user interface metaphor independent" applications.

The UIML description of an interface is completely independent of any specific platform. UIML is based on XML and is itself a meta-language, since it almost has no pre-defined *tags*, and the ones that it has are just for describing the abstract structure of the interface, but not the concrete interaction objects [Fer05, Ope04, PE01].

For rendering a UIML GUI description, it is necessary to translate the UIML "model" to an existing programming language or to a renderable UI description language (e.g. XUL, XAML).

XAML - eXtensible Application Markup Language

XAML (pronounced *zamel*) is Microsoft's new XML-based presentation layer declarative markup language. It separates "the UI definition from the run-time logic by using code-behind files, joined to the markup through partial class definitions" [MSDb].

2.9 Summary

This chapter surveyed a number of concepts and definitions needed throughout the rest of the document. In particular, the concepts of software process and software engineering method have been distinguished. Furthermore, the concepts of software model and Design by Contract have been addressed, and the purpose of software modeling in the scope of model-driven development has been clarified. Also the types of models in MDD, and the model transformation processes, techniques and strategies have been broached.

[2] http://www.mozilla.org

[3] http://www.harmonia.com and http://www.uiml.org

[4] http://www.oasis-open.org/

In addition, topics required for the definition of the metamodels presented in chapter 4 were discussed, namely the relevant parts of the UML metamodel, including OCL and action semantics, task modeling techniques and a notation for representing abstract presentation models. Besides that, the package dependency "merge" relation has been explained.

Finally, a short survey of XML-based UI description languages has been accomplished.

Chapter 3

State of the Art

This chapter presents a state of the art survey on UI automatic generation, and analyses the current approaches through the definition of a comparison framework.

3.1 Introduction

This chapter briefly surveys the most representative approaches for user interface generation from system models and compares and analyses them from a model-driven development perspective, according to a set of described properties. Some non-model-driven approaches are also surveyed for discussing their simplicity and ease of use and comparing them to the other approaches.

As we saw earlier, typical methodologies for modeling interactive applications use disparate views, or sub-models, to capture different aspects of the system (domain or application model, task model, dialogue model, abstract and concrete presentation models) [Pin00, Nun01].

Most of the existing approaches to UI generation demand the specification of a UI model. Typical models for the model-based development of interactive systems are identified in section 3.2 and were surveyed by Pinheiro da Silva [Pin00].

Some research has been made in order to model interactive systems using UML diagrams, but it also involves the full specification of the user interface [Pin02].

As mentioned in chapter 1, a typical approach to software engineering using UML starts by developing a sketch of the core system model by producing a structural or domain model, which models the system's domain classes, its attributes, relations and methods, and a functional or use case model, which models the user's intended operations to be accomplished on the system through its user interface. An important achievement would be to be able to generate a UI model from the domain and use case models, in such a way that from this set of models an interactive application prototype could be automatically generated. This way, the generated application or application prototype would be completely consistent with the system models and would be specially suited to be used for validating the system model with the end users, and for further refinements until

a final application is obtained.

User interface generation approaches may be categorized as follows:

- Generation of final UI executable code from a model view of the user interface taken from the system model;
- Generation of UI abstract model from the structural or behavioral views of the system model. To be able to execute these user interfaces, the UI abstract model must be subject to a model to code transformation, or it must be interpreted in an appropriate simulation environment.

Other possible categorizations of UI generation approaches, are *code generation* vs *simulation environment*, *just UI generation or prototyping* vs *completely functional interactive application generation*, or *model-driven generation process* vs *non-model-driven generation process*.

We intend as model-driven an approach that comprises the definition of models, conforming to pre-existing or constructed metamodels, model-to-model and model-to-code transformation processes. This way, a model-based approach may or not be model-driven, depending on the existence of those aspects.

The remaining of this chapter is organized as follows:

- In the next section, model-based user interface environments, and the kinds of models that are typically used in those environments, will be addressed.
- In section 3.3, model-based UI generation approaches are surveyed, distinguishing the approaches that follow a model-driven paradigm from the other approaches.
- In section 3.4, other UI generation approaches are addressed.
- Analysis and discussion of the surveyed approaches is taken care of in section 3.5, and a list of enhancement opportunities, identified in the surveyed approaches, is presented.
- Finally, section 3.6 concludes the chapter.

3.2 Model-based user interface development

In this section, the models used for model-based user interface development will be surveyed. Some of those models may be used in a model-driven user interface development process, as long as they are complete and rigorous.

Model-based development techniques of user interfaces construct a more or less declarative User Interface Model (UIM) that is typically composed of various sub-models, or model views. This UIM captures the relevant aspects of the UI and will hopefully serve for obtaining an executable or, at least, directly compilable,

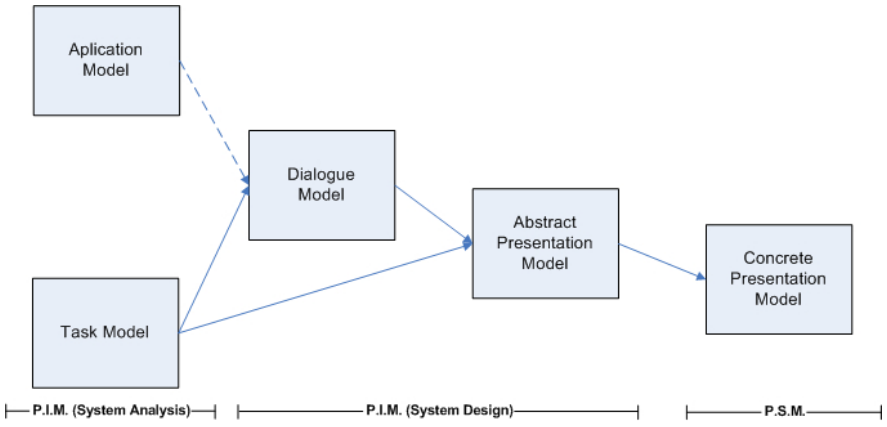

Figure 3.1: Model-driven development of user interfaces

concrete user interface. Nevertheless, there are few cases of automatic generation of User Interface implementations from UIMs [Pin00].

UIMs are typically developed using a model-based user interface development environment (MB-UIDE). Different MB-UIDEs use different kinds of models specified with different kinds of modeling languages.

Typically, a model-based UI development process begins with the construction of a task model and sometimes a dialogue model. Afterwards, an abstract interaction model (or abstract presentation model) is built and at the end of the process a concrete interaction model (or concrete presentation model) is constructed. All these submodels form the User Interface Model (UIM), that is a view or submodel of the system model. Figure 3.1 shows this relation between submodels. The construction of these sub-models may use a diagrammatic notation, such as State Transition Networks, Petri Nets or UML, or a textual specification, such as grammar-based notations or formal methods languages. It can also use a XML-based UI description language to create declarative descriptions of the UI.

There are several kinds of MB-UIDEs and each may use a different notation for UI modeling or even different sub-models for specifying the UIM, or for capturing different characteristics of the UI. Pinheiro da Silva [Pin00, Pin02] defines a framework for describing MB-UIDEs, and uses the framework for analyzing 14 MB-UIDEs. Pinheiro da Silva also addresses the ability of UML for modeling the UI of interactive systems [PP01].

User Interface Models provide a description of the UI at different levels of abstraction. Platform dependent user interface models make use of widgets and functionality that may be specific to one given platform. Platform independent declarative User Interface Models provide an abstract description of the UI, that can be reused and can be refined to more concrete (platform dependent) models. UIMs can, then, be found at different levels of abstraction during the UI design process. A UIM provides an infrastructure for allowing automated tasks in the

UI design and implementation processes.

Looking at these user interface models from a model-driven development perspective (fig. 3.1) one can consider application models and task models to be platform independent models (P.I.M.). Dialogue models and abstract presentation models are system design models in a platform independent manner. Concrete presentation models are platform specific models that may be subject to direct code generation.

MB-UIDE	Application Model	Task-Dialogue Model	Presentation Model	
			Abstract	Concrete
ADEPT	Problem Domain	Task Model	Abstract UI Model	Prototype Interface
AME	Application Domain	OOD	OOA	Prototype
FUSE	Problem Domain Model	Task Model	Logical UI	UI
HUMANOID	Application Semantics Design	Manipulation, Sequencing, Action side effects	Presentation	Presentation
ITS	Data Pool	Control Specification in Dialog	Frame Specification in Dialog	Style Specification
MASTERMIND	Application Model	Task Model	(none)	Presentation Model
MECANO	Domain Model	User Task Model / Dialog Model	(none)	Presentation Model
TADEUS	Problem Domain Model	Task Model / Navigation Dialogue	Processing Dialogue	Processing Dialogue
TEALLACH	Domain Model	Task Model	Presentation Model	Presentation Model
TRIDENT	Application Model	Task Model	(not surveyed)	Presentation Model

Table 3.1: MB-UIDEs submodels (borrowed from [Pin00])

As it was previously written, UIMs are declarative models composed of one or more sub-models, each capturing a relevant part of the system. In his survey, Pinheiro da Silva shows which sub-models compose a UIM within the studied MB-UIDEs (see table 3.1).

Typical UIM sub-models, identified by Pinheiro da Silva [Pin00] and Traetteberg [Træ02], are:

Abstract Presentation Model - describes the structure of the visual parts of the user interface, in terms of Abstract Interaction Objects (AIO).

Concrete Presentation Model - describes in detail the visual parts of the user interface, in terms of Concrete Interaction Objects (CIO) or widgets.

Task Model - describes the tasks and relations between tasks that users may perform when interacting with the application.

Dialogue Model - describes the actions or events that the user is allowed to perform on the UI and the events that the system can perform at the UI when responding to the user.

Application Model - describes the properties of the application relevant to the UI, typically its static structure and behavior interface.

In his survey, Pinheiro da Silva considers Task and Dialogue models as belonging to a same class of models that address task decomposability until a degree that allows the mapping of user tasks to events on the system.

Some MB-UIDEs, specially those concerning user adaptive UIs, also take advantage of a User or Usage Model. Others, specially those concerning platform adaptive UIs or ubiquitous applications, may take advantage of Platform or Environment models [Træ02]:

User Model/Usage Model - describe characteristics, abilities and preferences of end-users when interacting to the system.

Platform or Environment Models - describe the characteristics of the device and of the cultural context of the interaction.

Each UIM submodel has its own constructs. Table 3.2 shows the constructs identified by Pinheiro da Silva for each kind of model view in the MB-UIDEs surveyed.

Model-based UI development processes are typically iterative and incremental, consisting of a sequence of UIM refinement steps towards a concrete UI implementation for some real or virtual platform. This is in line with MDA objectives of model transformation and traceability from the more abstract (platform independent) models to the more concrete (platform specific) models.

MB-UIDEs usually provide a graphical environment that aids in the construction of the UIM. Some MB-UIDEs provide a graphical editor for editting diagrammatic model views, or submodels. Others provide an editor for textual UIMs, though.

Pinheiro da Silva [Pin02] divides the UI development process into UI design process and UI implementation process. Because the design process is typically incremental, MB-UIDEs usually provide means for refining UIM, thus supporting models at different levels of abstraction. Some MB-UIDEs also provide a design

Submodel	Constructor	Function
Application Model	Class	An object type defined in terms of attributes, operations and relationships.
	Attribute	A property of the thing modelled by the objects of a class.
	Operation	A service provided by the objects of a class.
	Relationship	A "connection" among classes.
Task-Dialogue Model	Task	An activity that changes the state of specific objects, leading to the achievement of a goal. Tasks can be defined at different levels of abstraction allowing the definition of sub-tasks.
	Goal	A state to be achieved by the execution of a task.
	Action	A concrete task that can be executed.
	Sequencing	The temporal order that sub-tasks and actions must respect for carrying out the related high-level tasks.
	Task pre-condition	Conditions in terms of object states that must be respected before the execution of a task or an action.
	Task post-condition	Conditions in terms of object states that must be respected after the execution of a task or an action.
Abstract Presentation Model	View	A collection of Abstract Interaction Objects (AIOs) logically grouped to deal with the inputs and outputs of a task.
	AIO	A user interface object without any graphical representation and independent of any environment.
Concrete Presentation Model	Window	A visible and manipulable representation of a view.
	CIO	A visible and manipulable user interface object that can be used to input/output information related to user's interactive tasks.
	Layout	Information for manual placement of CIOs in windows, or an algorithm that provides the automatic placement of CIOs in windows.

Table 3.2: UIM submodels' constructs (adapted from [Pin00])

assistant that checks the model according to a set of design guidelines. MB-UIDEs that support UI implementation, allow for the refinement of the model until a concrete UI model is obtained. Sometimes, a UI implementation can be generated. The support for automatic generation of code is, indeed, one of the main advantages of model driven development.

The use of a User Interface Management System (UIMS) is one of the solutions for running concrete UI models. Another solution is the generation of final compilable code from the UI model. Betts *et al.*[BBF+87] presents a summary of the characteristics that shoud be found in a UIMS system and a contribution to the development of a taxonomy of a UIMS. These characteristics are, from the point of view of the developer, the support for consistent user/application dialogues definition, the imposition of external control on the application provid-

ing support for a taillored and extensible presentation of the applications' output and the inclusion of an interactive component that supports the interaction between the application and a range, from novice to experts, of end users. From the point of view of the end users, the primary goal of a UIMS is to support the easy and effective use of an application, contributing for a consistent UI across applications, multiple levels of help and trainning support, ease of extensibility of the application's UI and the end user UI tailloring cappability.

It is also addressed in [BBF+87] how a UIMS could support each of the characteristics it should have.

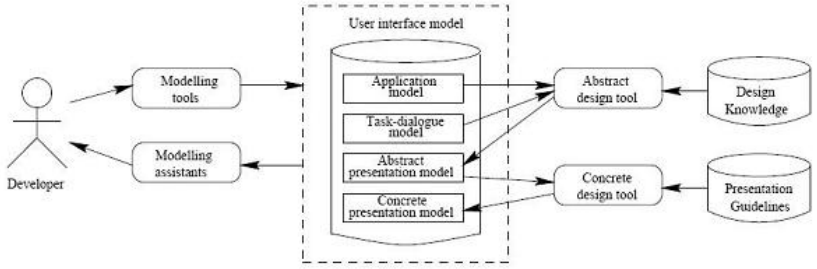

Figure 3.2: The user interface design in a MB-UIDE (borrowed from [Pin00])

In [Pin00] an approach to UI design automation is sketched in two steps (see figure 3.2). First, an abstract design tool generates an abstract presentation model from the application model and/or the task-dialogue model. In this process a design knowledge database may be used to supply information that can control the options that have to be made. Then, a concrete design tool generates a concrete presentation model from the abstract one, in which process a design guideline database may be used.

In what respects to the UI implementation process, Pinheiro da Silva [Pin00] sketches three approaches to generating and executing a user interface from a UIM, represented in figure 3.3. In the first approach (figure 3.3a), the MB-UIDE generates the UI source code that will be used for the UI layer of the application. In the second approach (figure 3.3b), the MB-UIDE generates a concrete UI specification that is animated by the UIMS engine. In the third approach (figure 3.3c), the MB-UIDE embbeds a UIMS, and executes the UI itself, from the concrete model it already has. In the first and second cases, one can talk about a UI generator. In the first case is generated a UI targeting a chosen programming language, while in the second case, a UIMS input concrete model is generated. In the second and third cases, one can talk about a UI runtime system.

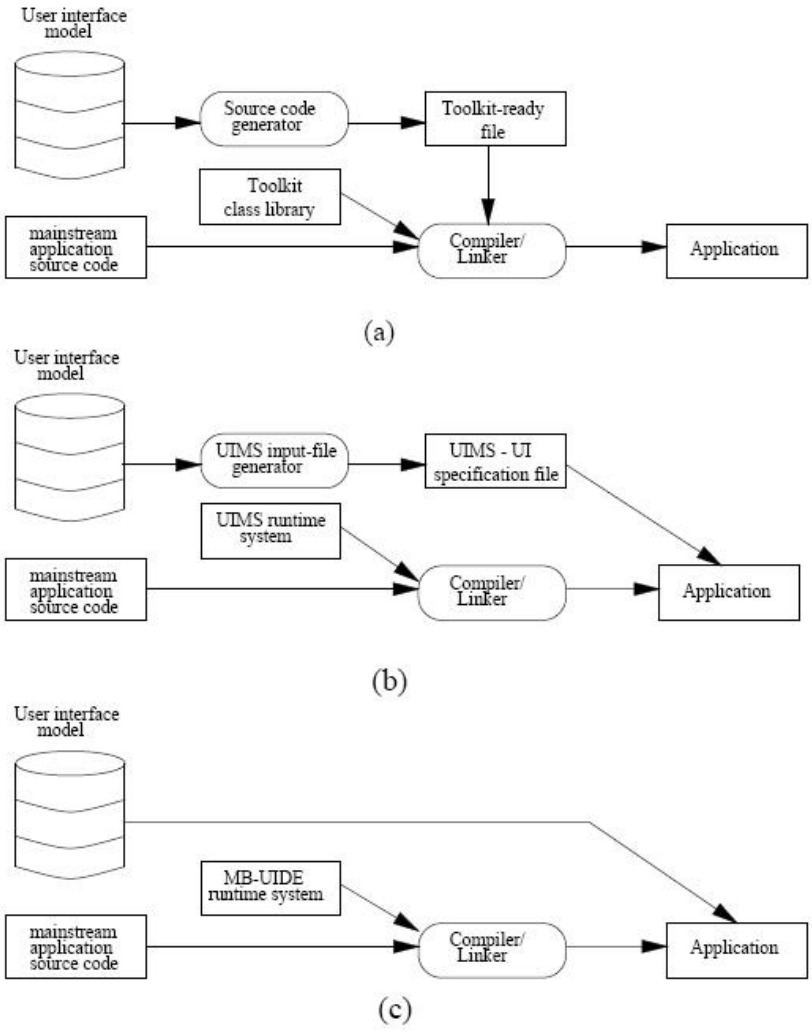

Figure 3.3: The user interface implementation in a MB-UIDE (borrowed from [Pin00])

3.3 User interface automatic generation

This section presents model-based UI generation approaches, distinguishing the approaches that follow a model-driven philosophy from the ones that don't.

3.3.1 The XIS approach

The most recent version of the XIS-UML Profile and approach [dSSSM07] has been developed within the ProjectIT (PIT) research project [SV08, Sil04], and is based on a prior version [Sil03] of XIS.

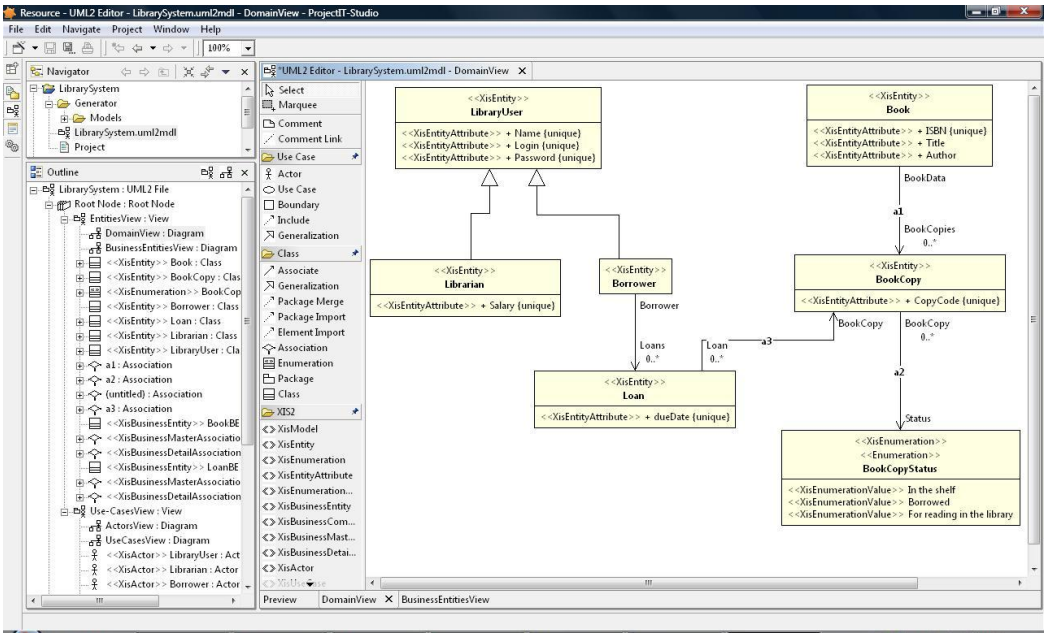

Figure 3.4: Xis domain model, edited in ProjectIT-Studio, for the LibrarySystem example.

ProjectIT is a research project from INESC-ID that aims the development of software tools for supporting model-driven software development processes. Two complementary tools have been developed within ProjectIT:

- **ProjectIT-Enterprise**, which is an integrated environment for CSCW (computer-supported cooperative work) [SV08].
- **ProjectIT-Studio**, which is a model-driven integrated development environment [SV08, SVS+06, SSFV07] that turns possible to approach a new software system's development by specifying user requirements using a requirements specification language (PIT-RSL). Then, using the XIS/UML profile [Sil03, dSSSM07], a UML model of the system may be constructed. Finally, using a template specification language (PIT-TSL), a set of scripts defining model to code (M2C) transformations may be written, in order to enable code generation from the specified XIS/UML model [SV08].

The research made towards the XIS approach lead to a multi-viewed organization of the concerns that shall be modeled in a software system. A XIS model shall have the following complimentary views [dSSSM07]:

- **Entities View**, which is in turn subdivided into a Domain View and a Business Entities View. The **Domain View** models the domain entities

by using a UML class model with classes (stereotyped with ≪XisEntity≫), associations and attributes (stereotyped with ≪XisEntityAttribute≫), and enumerations (classes stereotyped with ≪XisEnumeration≫ that extensively list a collection of possible values, enumeration attributes, stereotyped with ≪XisEnumerationValue≫). Figure 3.4 shows an example of a Xis domain model edited in ProjectIT-Studio.

The **Business Entities View** is used to group together a set of domain entities (≪XisEntity≫), in a coarser granularity entity (≪XisBusinessEntity≫) that shall be manipulated in the context of a use case. A business entity may be specified by designating a master entity and a sequence of detail entities. A business entity may also be specified by defining 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 **Use-Cases View**, which relates the actors with the use cases that each actor can perform. The UseCases View also defines the relationship between each use case (≪XisUseCase≫) and the business entities upon which the actors related to that use case can perform operations (≪XisOperatesOnAssociation≫). The ≪XisOperatesOnAssociation≫ stereotype has a tagged-value, *operations*, that enables the definition of the set of operations that can be performed in the business entity - subset of the operations listed in the ≪XisBusinessMaster≫ association.

- **User-Interfaces View**, which defines an user interface abstract presentation model, and is subdivided in the **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 model may, then, be inputted to a model to code (M2C) generation process that, in ProjectIT, is made available through templates. Figure 3.5 shows the architecture of the projectIT approach to MDD, and figure 3.6 shows the multi-viewed organization of XIS models.

The smart approach illustrated in figure 3.6, in which the UI view will be automatically obtained from the Entities and the Use-Cases views, is not yet made available in the ProjectIT/Studio.

ProjectIT/Studio allows the definition of model-to-text (M2T) or M2C scripts that transform XIS stereotyped models into documents or code files that may be targeted to any architectural layer of the final application (eg.: Data layer, Business Logic Layer, User Interface Layer). The XIS UML profile was thought, in its first version [Sil03], to support a Model-View-Controller (MVC) architectural pattern. The most recent version of XIS is somewhat closer to the Boundary-Entity-Control (BEC) architectural pattern [dSSSM07], where entities are the

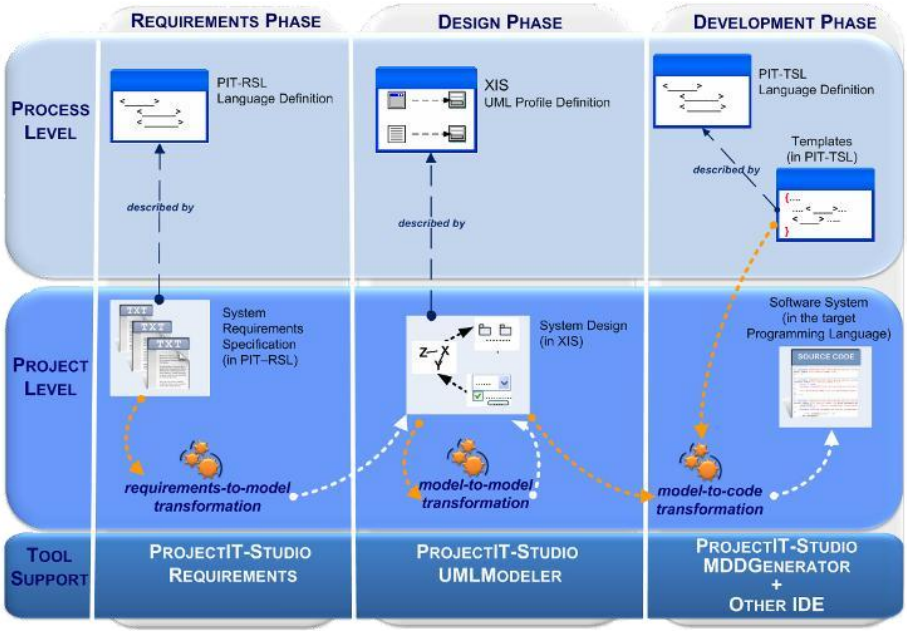

Figure 3.5: The ProjectIT tool support to MDD (taken from http://isg.inesc-id.pt/alb/ProjectIT-Studio@79.aspx).

information that is to be persisted in the system (Entity), business entities aggregate entities in directly manipulatable objects, operated by actors in the context of a use case (Control), and Interaction Spaces, through which the actors interact with the system within a use case, have the role of a Boundary.

All model views in XIS are platform independent, and M2C scripts operate on XIS models. XIS and the ProjectIT tools allow the generation of models from models - that is the case of the User-Interfaces View in the smart approach, although it isn't available yet in the ProjectIT-Studio tool. The XIS profile doesn't support OCL nor the full specification of operations – it only allows the declaration of operations' name, not their signature –, nor semantics (body or pre-/post-conditions) [dSSdS08].

3.3.2 The Wisdom approach

The Wisdom method [Nun01] is a user-centered object-oriented (UC-OO) method for the model-based development of interactive applications. In [Nun01] Jardim Nunes proposes the Wisdom Lightweight software engineering method with three major components:

- The Wisdom Process - A user-centered evolutionary and rapid prototyping process model, specifically adapted for small teams of developers.

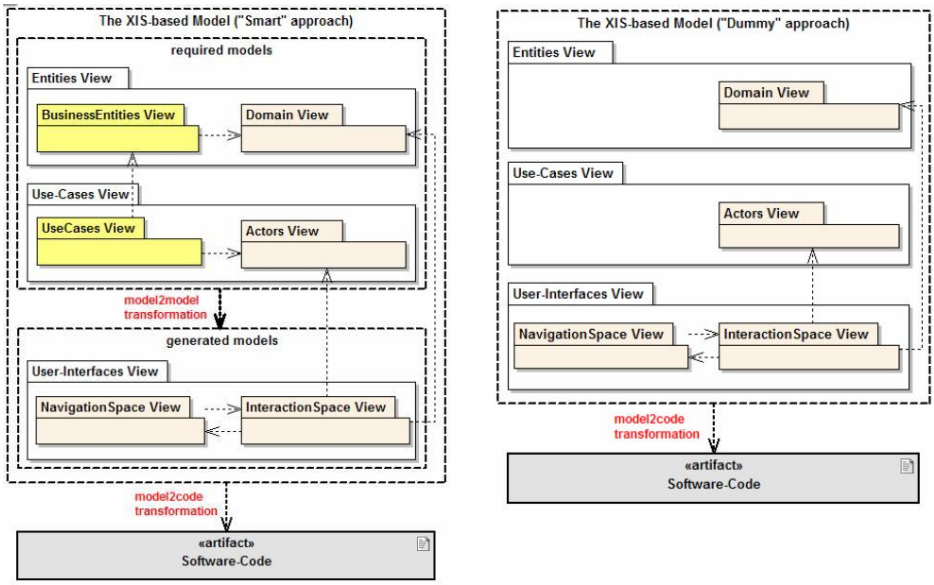

Figure 3.6: The XIS design approaches to interactive systems generation [dSSSM07].

- The Wisdom model architecture - A set of UML models that support the different process development phases involved in the Wisdom process.

- The Wisdom notation - A set of modeling notations based on a subset of the UML and a UML profile that extends UML for better modeling interactive systems.

The Wisdom method is driven by essential use cases and task flows. Essential use cases, as proposed by Constantine and Lockwood [CL99, Con06a, Con06b, CWNL03, Con95], differ from the standard UML use cases [RJB99] in the sense that the former corresponds to top-level tasks, that is goals or external tasks that the user wishes to achieve, and the latter corresponds to a pack of functionality that the system provides to its users [RJB99, Nun01].

In Wisdom, each essential use case is detailed using activity diagrams. To avoid a scenario explosion in complex use-cases, activity diagrams with control conditions (e.g.: if) are used to model all scenarios of each essential use case in only one activity diagram.

Essential use-case diagrams, with each use case detailed by an activity diagram, forms the use case model. The use case model, together with the domain model and the user role model, shall be developed in the requirements process phase (see figure 3.7).

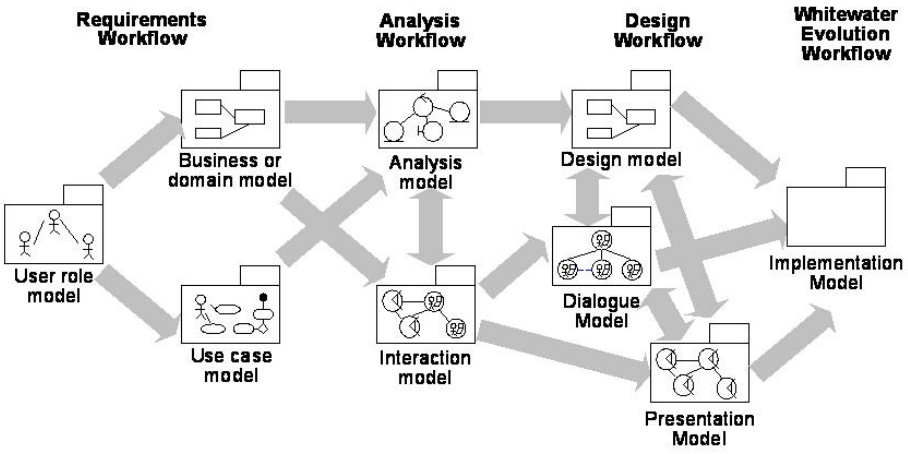

Figure 3.7: The Wisdom model architecture [Nun01].

In the analysis phase, an analysis model models the internal system architecture, stating how different analysis classes participate in the realization of different use cases. Analysis classes may be stereotyped as «Boundary», «Entity» or «Control». Wisdom's boundary analysis classes model the interaction of the system to non-human actors (external systems). An interaction model is also built in this phase. This model structures the user interface, by identifying the different elements that compose the dialogue and presentation structure of the system, and how they relate to the domain specific information in the functional core [Nun01]. An interaction model models the UI architecture design, by using variants of the previous analysis stereotypes, namely «Entity», «Task» and «InteractionSpace». Wisdom's interaction spaces model the interaction of the system to human actors. The analysis model and the interaction model form the Wisdom UI architecture model [Nun01].

In the design phase, a design model may refine some issues of the analysis model, and a dialogue and a presentation models are built. The dialogue model specifies the dialogue structure of the interactive application and the presentation model defines the physical realization of the perceivable part of the interactive system, namely the defined interaction spaces [Nun01].

Although not having an automatic generation tool, yet, Wisdom model development may be accomplished using WinSketch[1], that is a tool for constructing and syntactically validate the set of model views that form the interactive application model, according to Wisdom.

This approach could be framed into a model-driven software development process, in which the final system UI would be generated from the models.

[1]http://apus.uma.pt/ winsketch

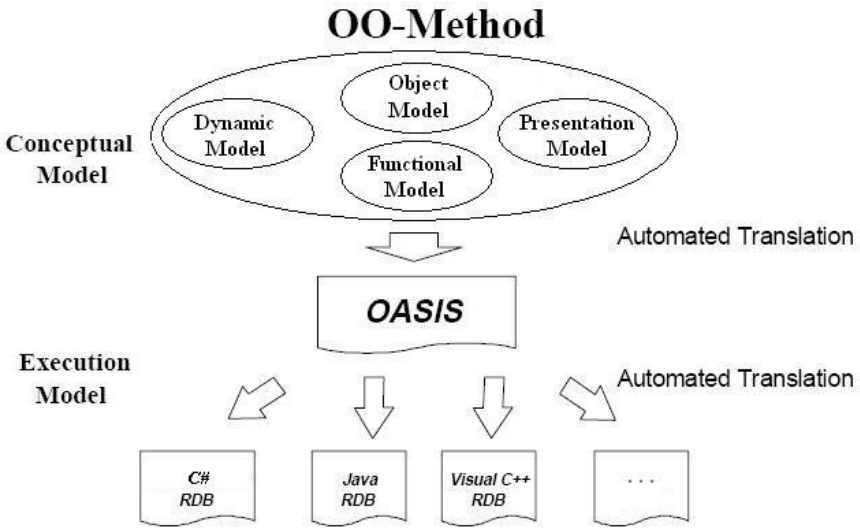

Figure 3.8: The OO-Method approach (adpated from [PIP+97, PI03]).

3.3.3 Olivanova and the OO-Method

Olivanova[2] is a suite of commercial tools for the production of business applications, based on three research academic and industrial works [PMI04, Mol04, PIP+97]:

OASIS is a formal textual language to rigorously specify object-oriented systems in a declarative manner. OASIS specifications are directly executable in declarative environments similar to Prolog.

OO-Method is an object-oriented method that combines diagrammatic modeling techniques (like UML) with the OASIS specifications.

Just-UI is a pattern language extension to the OO-Method that allows the specification of UI models. Just-UI also defines the mappings from abstract UI specifications to concrete implementations for different devices.

The OO-Method approach (see figure 3.8) 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 OO-Method 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 modeling patterns provided by the method, to an OASIS specification - the execution model.

[2]http://www.care-t.com

The OO-Method starts, then, with the construction of a conceptual model, which is in turn composed of the following sub-models [PIP+97, PI03]:

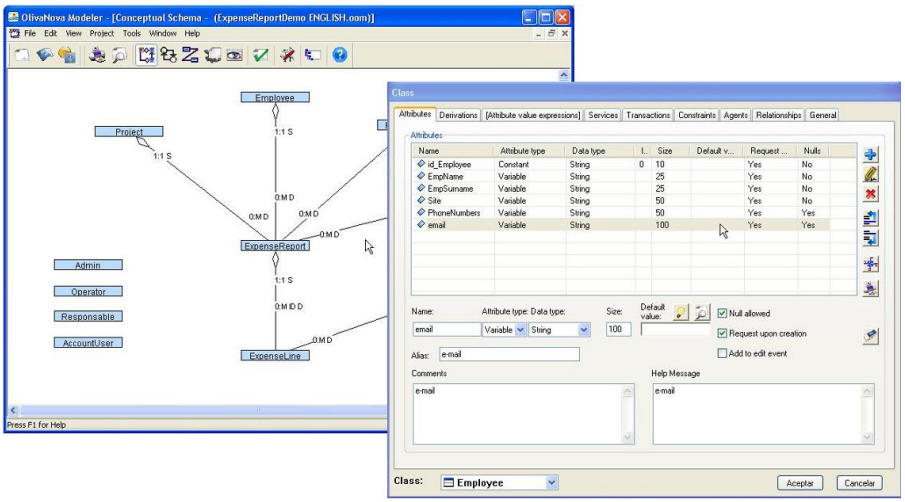

Figure 3.9: Olivanova object model editor (taken from http://www.care-t.com).

- **Object Model**. - Is represented through a UML class diagram, capturing domain classes and classes associated to user roles. Figure 3.9 shows Olivanova object model editor. 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). Inheritance relations may be permanent or temporary. The former case is the normal semantics of object-oriented inheritance; in the latter, it corresponds to a role that is temporarily played by one class, and may be activated/de-activated by services or conditions.

  A class service (operation) may be private or shared (public), and creation and destruction services are marked, in order to infer their behavior.

- **Dynamic Model**. - Is used to specify valid object lifecycles and inter-objectual interaction. To specify valid object lifecycles, a state transition diagram is used per class. Each state transition diagram represents the valid states and the valid transitions between states of objects of a class. Transitions may have attached control (guard) 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. With global interactions it is possible to declare interobjectual transactions.

- **Functional Model**. - The 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 requiring a formal methods knowledge by the software engineer, 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.

  There are three types of attributes [PIP+97, PI03]:

  - **Push-pop** attributes - Attributes whose value is increased or decreased, by a given value, by relevant services. Services that reset the attribute for a given value may also exist.
  - **State-independent** attributes - Its value depends only on the latest action that has occurred.
  - **Discrete-domain valued** attributes - Attributes that take their value from a limited domain. The different values of this domain define the valid situations that are possible for objects of the class. Through the activation of a carrier action (that assigns a domain value to the attribute) the object reaches a specific situation. That situation is abandoned when another event occurs (a "liberator" event).

- **Presentation Model**. - The last step is to specify how users will interact with the system [PI03]. 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 [MPM+01, MH03]. The presentation model of the OO-Method comprises an action hierarchy tree, and a set of interaction units that may have elementary UI patterns. There are four kinds of interaction units, namely a service interaction unit, an instance interaction unit, a populating interaction unit and a master/detail interaction unit. UI patterns allow to limit and constrain the interaction units. Examples of UI patterns are the pattern of introduction, of defined selection, or of state recovery, amongst others [Mor03]. The Presentation model is defined, within the Olivanova modeler tool, by filling forms and selecting available interaction units and patterns.

The abstract execution model is based on the concept of conceptual modeling patterns. The OASIS specification model must accurately state the implementa-

tion-dependent features associated to the selected object society machine representation. The OlivaNova transformation engines provide a well-defined software representation of the conceptual modeling patterns in the solution space.

3.3.4 Elkoutbi et al. approach - Use cases formalized by collaboration diagrams

Elkoutbi et al. [EKK06] approach UI generation by identifying usage scenarios. Their approach starts from a system structural model, including domain classes and interface classes, with OCL constraints, and a use case model. A use case is intended as a generic description of an entire transaction, and is formalized by a set of UML collaboration diagrams, each corresponding to a use case scenario. Each scenario (collaboration diagram) is, then, classified by type (whether it is normal or exceptional) and by frequency of use (how often it is likely to occur). Each collaboration diagram message is manually labeled with UI constraints (inputData and outputData) that identify the input and output message parameters for the UI. From UI constraints it will then automatically produce message constraints with UI widget information.

In the next step, a transformation algorithm is applied to each UI labeled collaboration diagram, resulting in a statechart diagram for each object in the collaboration diagram. These state diagrams are called partial specifications, and will need to be integrated into a single statechart for each class/use-case pair.

Then, the partial specifications must be analysed in order to give common labels to equivalent states in different partial specifications. State labeling and statechart integration are done incrementally, until there is a single state diagram per class/use-case pair.

Also, some state diagrams are integrated across use cases enabling subsequent design and implementation.

Elkoutbi's approach is then able to derive UI prototypes for every interface object defined in the structural model. For each interface object, a standalone prototype is generated from all of its state diagrams. Each prototype contains a menu to switch between the different use cases.

Final prototype generation is obtained by generating a graph of transitions for each interface object in a given use case, in which graph nodes represent transitions in the state diagram and edges represent precedence of execution between transitions. Non-interactive transitions are then masked (removed) and new edges are created to "bridge" the removed transition nodes. Transition nodes are grouped together according to a set of rules [KEK01] in order to create a new graph in which the nodes represent user interface blocks (UIB).

UIBs are then combined in order to obtain more complex blocks, according to a set of defined rules [KEK01], trying to concentrate in the minimum number of UIBs the scenarios that have the highest frequency. For each final UIB a graphic

frame is generated, containing all the widgets of all the transitions belonging to the concerned UIB.

The dynamic aspect of the UI is controlled by the behavior specification (state diagram) of the underlying UI object.

Running the generated prototype means symbolic execution of the state diagram, that is equivalent to traversal of the transition graph with only interactive transitions. The prototype responds to all user interaction events captured in the graph and ignores all other events.

When running the prototype, every time the execution reaches a node in the graph from which several paths are possible, the prototype displays a scenario selection box.

3.3.5 Martínez et al. approach - Use cases formalized by UI enriched sequence diagrams

In [MESP02], Martínez et al. present a methodology for deriving UIs from early requirements existing in an organization's business process model. Their approach aims at generating a UI that can be validated by organizational users as early as possible.

The approach has three phases. The first phase starts with the informal requirements of the organizational environment and consists on the construction of a business process model (step 1 in figure 3.10). As a result, a strategic dependency model and a strategic rationale model, which reflect the business process of the organization, are obtained.

The documentation from the first phase serves as input to the use case generation phase. This second phase, which also has only one step, follows a set of heuristics to detect an initial use case model (use cases and actors) as well as the normal interaction scenarios for each use case (step 2), which are represented using message sequence charts.

The third phase has five steps. It starts with scenarios completion, by adding exceptional or alternative scenarios for each use case message sequence chart (step 3). Each message sequence chart is then enriched with UI related information, in a use case synthesis step (step 4). The enriched message sequence charts are used in the generation of graphic components of the interface (step 5). In this step, application forms for the interface objects present in the sequence diagrams are generated, in a target language. The message sequence charts are also used to generate state transition diagrams for the interface objects and control objects present in the sequence diagrams (step 6), forming a navigation model.

Finally (step 7), the generated UI is animated by symbolically interpreting the generated forms and the transition diagrams.

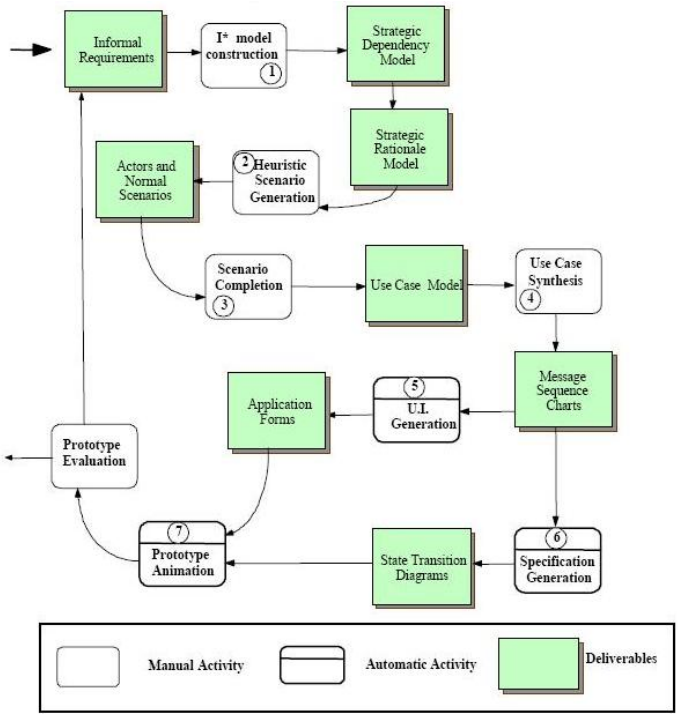

Figure 3.10: Martínez *et al.* method for UI generation (taken from [MESP02]).

3.3.6 Forbrig et al. approach - Pattern-driven model-based UI generation

Forbrig et al. [WFDR05, WFR05, RFSS07, JSS+07, FDRS04, RFD04] 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 (see figure 3.11). Then, from the previous models, a set of selectable patterns is identified enabling its selection by the modeler 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 [WFR05] and PIM Tool ("Patterns in Modeling" tool) [RFSS07] enable this computer assisted approach.

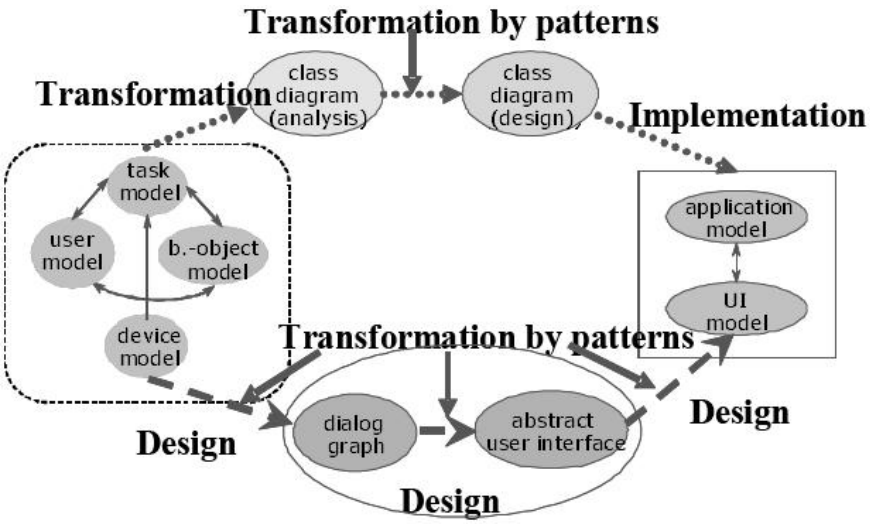

Figure 3.11: Pattern-driven model-based UI development approach (taken from [WFDR05, WFR05]).

ZOOM-M example

```
typedef FriendList = List[Contact];
public struct User {
  invariant { userID != null }
  public String userID;
  public UserType userType;
  protected String password;
  public boolean loggedIn = false;
  public User(String userID)
    requires { userID != null }
    ensures { this'.userID == userID };
  public void login(String maskedPassword)
    ensures {
      this'.loggedIn ==
       (encrpt(this.password) == maskedPassword)
    };
}
```

ZOOM-FSM example

```
fsm Main() {
  state NotLogin;
  state Login { fsm LoginFSM() { ... } };
  state Idle { fsm IdleFSM() { ... } };
  transition initial to NotLogin;
  transition NotLogin to Login : evLogin;
  transition NotLogin to finalState: evExit;
  transition Login to NotLogin : evLoginFail;
  transition Login to Idle : evLoginSuccess;
  transition Idle to NotLogin : evLogout;
}
```

ZOOM-UIDL example

```
<Window id="String" name="Add Contact"
     show="true">
  <Panel>
    <Label text="UserID"/>
    <TextBox name="userID" type="text"
       editable="true" columns="2"/>
    <Label text="FirstName"/>
    <TextBox name="firstName" type="text"
       editable="true" columns="2"/>
    <Label text="LastName"/>
    <TextBox name="lastName" type="text"
       editable="true" columns="2"/>
    <Label text="EMail"/>
    <TextBox name="email" type="text"
       editable="true" columns="2"/>
    <Button name="addButton" text="Add"/>
    <Button name="cancelButton" text="Cancel"/>
  </Panel>
</Window>
```

Figure 3.12: Examples of ZOOM models textual representation (adapted from [JSL+05]).

3.3.7 The ZOOM project

The ZOOM approach to interactive systems modeling and development [JSL+05] provides a set of processes, notations, and supporting tools that enable model-driven development. ZOOM, which stands for *Z-based OO modeling* notation, is an object-oriented (OO) extension to the formal specification language Z.

ZOOM separates an application into three parts - structure, behavior, and user-interface - and provides three separate, but related, notations to describe each of those parts: ZOOM-M for structural models; ZOOM-FSM for behavioral models; and, ZOOM-UIDL for UI models. ZOOM provides a Java-like textual syntax for structural and behavioral models and an XML-based language for the User-Interface model (see figure 3.12). Furthermore, ZOOM provides a graphical representation of models consistent with UML diagrams (see figure 3.13) [JSQ+07, JSL+05]. This enables a graphical formal modeling of a software system.

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

Furthermore, 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 an UI generated from the UI model. This "compilation" process is more complex than simple language compilation, though. The generated code must, not only meet all functional requirements, but the generation process must address the choice of architecture, data structures and algorithms [JSL+05, JSQ+07]. Figure 3.14 shows an overview of the ZOOM approach to MDD.

To support decisions related to the choice of architecture, data structures

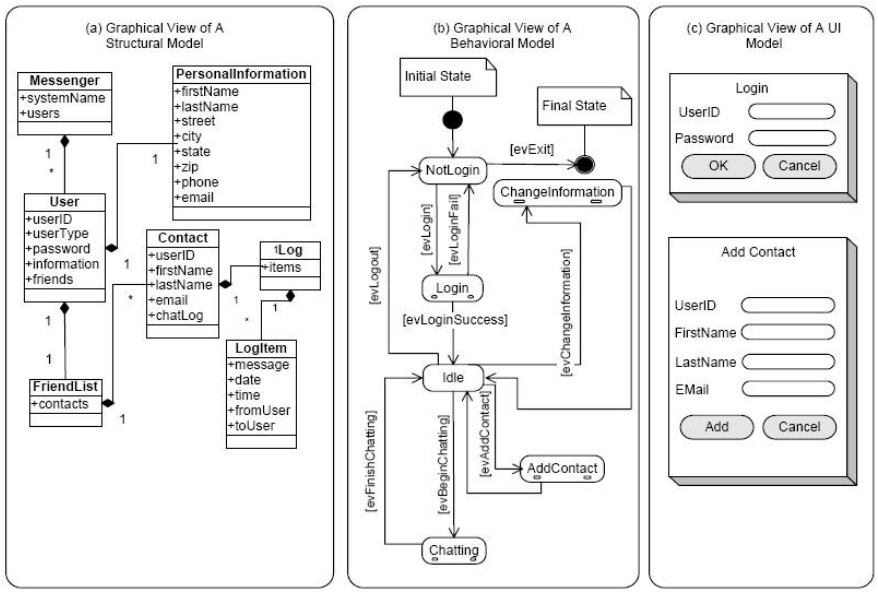

Figure 3.13: Examples of ZOOM models graphical representation (taken from [JSL+05]).

and algorithms, the ZOOM approach attempts to capture software architecture and design knowledge of human experts in a knowled base, which is used by knowledge-based compilation tools to carry out fully automated model compilation (code generation).

3.4 Other UI generation approaches

Many other approaches to UI generation exist. In this section, five non-model based approaches are surveyed.

3.4.1 The editing model of interaction

Dewan and Solomon [DS90] present an approach to suport the automatic generation of User Interfaces based on the editing model of interaction. In this model, data objects are edited using a generic data objects editor. Objects must have an interface that allows the use of the editing model of interaction. For instance they must implement methods for persistence (load and save methods). The concept of a "presentation" of the object, allows for the separation between the data in the object and its visual representation. An object may have several presentations, each one displaying a different view of the object.

So, in the editing model of interaction, a user interacts with an object by loading its presentations into a generic editor. The generic editor is bounded by

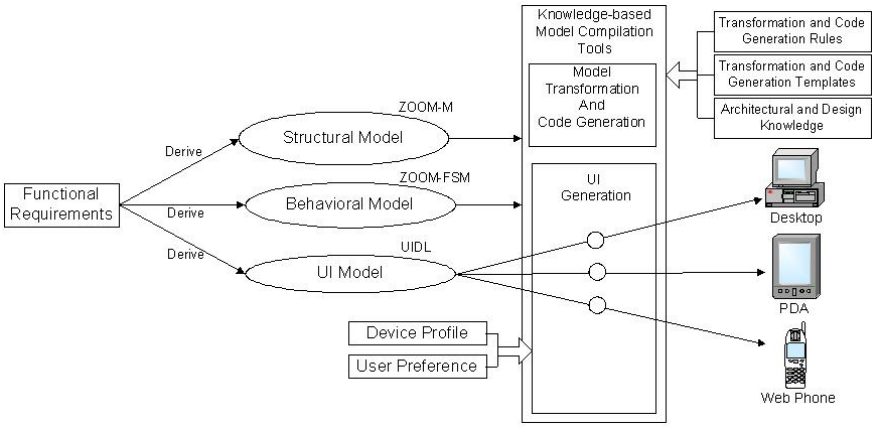

Figure 3.14: Overview of ZOOM (taken from http://se.cs.depaul.edu/ise/zoom/zoom.html).

the object loaded. The user proceeds by executing generic editing commands from the editor, that the editor passes to the object being edited through the editing supportive interface that the object must implement, comprising load and save methods. For less generic operations, the object may provide a set of methods that the editor allows the user to execute. This generic editor is a kind of UIMS for the editing model of interaction. The editing system is composed of a dialogue manager that supports the editing model of interaction by displaying the different presentations of an object in different subwindows of a window. If no presentation is defined for an object, the dialogue manager constructs a default presentation based on the types of the object's variables, defined in its class.

The editing model of interaction for generic objects may be compared to the use of syntax directed editors for editing a programming language, or any other language defined by a concrete grammar, such as synthesizer generator[3].

3.4.2 Using generic functional programming techniques

Through a few examples, Draheim *et al.* [DLW05] present Genoupe, a language that extends C# with new constructs for parametric polymorphism. Genoupe is a sort of precompiler language that allows for the definition of generative components. These generative parametric components are C# program generators. Draheim *et al.* present a few examples of using these program generators; amongst is the possibility of defining a C# Form generator for any class.

[AvEPvW04] uses generic functional programming techniques to create GUIs in an abstract and compositional way, using type-directed and statically typed higher-order graphical editor components (GECs).

[3]http://www.grammatech.com/products/sg/overview.html

3.4.3 Generating UI from XML Schema instances

Lay and Lüttringhaus-Kappel [LLK04] present a method for generating XML based UI languages based on the transformation of XML Schema instances through the use of XSLT stylesheets. The paper focuses, in particular, the generation of XML serialized Java Swing objects. These serialized objects are, then, part of a GUI application that implements the Model-View-Controller (MVC) architectural pattern, where the serialized UI objects play the role of the View, and the Model and the Controller are the XML Schema and the control structures, respectively. Depending on the data types defined in the XML Schema, the system uses specialized input fields when generating the GUI. When in the presence of a complex type, the system produces consecutive elements, if a sequence of fields is found, and produces additional GUI elements if alternative or repetitive elements are found. Lay and Lüttringhaus-Kappel point out the reason that the forms use a static layout of input elements, unless JavaScript is used, for not producing a real XML-based GUI language.

3.4.4 Adaptive Object Model

An adaptive object model (AOM) [YJ02] is a system that represents classes, attributes, relationships and behavior as metadata, stored in a database or XML files. Changes in the metadata (object model) reflect changes in the domain, and modify the system's behavior. An AOM architecture, or metamodel (see figure 3.15), is usually made up of several smaller patterns [YJ02], like for instance:

- **TypeObject** - allows the dynamic definition of new business entities for the system. TypeObject is used to separate an Entity from an EntityType. EntityType will have a set of shared attributes for a group of entities. It allows to represent a "is a" relation between business entities, i.e., it allows the specification of direct inheritance.

- **Property** - represents an entity attribute.

- **Entity-Relationship**. - Relationships are properties that refer to other entities, and are usually two-way associations. One of the ways to separate entity attributes from associations is by making two subclasses of Property: Attribute and Association.

- **Strategies and Rule Objects**. - A strategy is an object that represents an algorithm. The strategy pattern defines a standard interface that can be implemented by different algorithms. An object behavior may be defined by one or more strategies. Each application of strategy leads to a different interface. Strategies are usually used in AOMs for implementing the operations on the methods. When more complex business rules are needed,

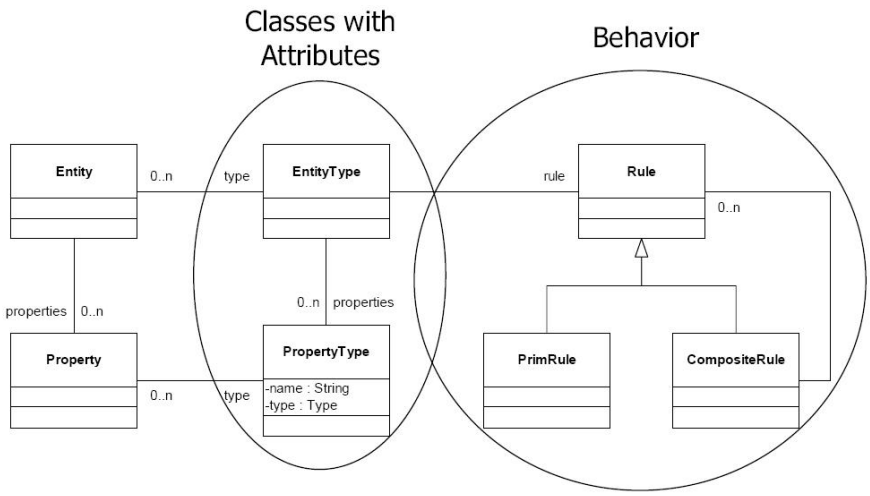

Figure 3.15: AOM typical meta-model structure

strategies can be combined using the Composite pattern. Rules that represent predicates are composed of conjunctions and disjunctions, rules that represent numerical values are composed of addition and subtraction rules, rules that represent sets are composed of union and intersection rules. These more complex Strategies are called RuleObjects [YJ02].

Figure 3.15 shows a typical AOM metamodel, where the application of the above mentioned patterns is illustrated. AOMs are present in this survey because they allow the execution of a system model. Of course an AOM system needs to be developed, and a user interface needs to be created. It is possible to extend EntityTypes to allow for some standard views and ease the process of building GUIs for AOMs.

3.4.5 Nguyen and Chun's approach to MDD

Nguyen and Chun's approach[4] to MDD [NC06] is based on the interactive modeling and animation of use case driven UML models.

The process begins by the manual construction of a use case model using the tool described in [NC06]. The use case model describes the system functions in terms of user goals, documenting the observable value the system provides to its users. Then, use cases are further elaborated by iteratively and interactively specifying use case steps, using a use case declarative specification language (see fig. 3.16), and then generating and executing sequence diagrams. From the use

[4]http://pnguyen.tigris.org/

```
create use case buy_product
  step 1 "Customer browses catalog to…"
  step 2 "System displays catalog"
  step 3 "Customer selects items to buy"
  step 4 "System acknowledges selection"
  step 5 "Customer goes to check out"
  step 6 "System displays check out screen …"
  step 7 "Customer fills in shipping …"
  step 8 "System calculates total …"
  step 9 "Customer fills in credit …"
  step 10 "System authorizes purchase …"
with
  actor "Customer"
  goal "Buy Product(s)"
  extension check_out at step 5
  extension authorize_purchase at step 9
```

Figure 3.16: Example of the Nguyen and Chun's use case specification language (taken from [NC06])

case steps, a sequence diagram may be derived, and objects collaborating in each step (or message) may be identified. From the sequence diagrams, the tool can also track the relationship between a use case and the domain objects, enabling the generation of a class diagram from the sequence diagram. It is possible to use the Groovy scripting language (see http://groovy.codehaus.org/) or Java to add behavior to classes' operations. That behavior is visible when executing the model.

This approach doesn't generate a UI for executing the software model, but it is worth mentioning here because it models the use cases by some formalized means and enables the execution of the UML model with the purpose to validate it and iteratively refine the model.

3.4.6 outSystems agile platform

OutSystems Service Studio, part of the outSystems agile platform, is a visual environment that enable developers to rapidly assemble and change web business applications. Service Studio enables the construction of web applications by specifying a user interactions flow that describes and keeps end-user navigation patterns across the application [Out]. Figure 3.17 shows this user interactions flow that resembles a dialog or a navigation model from MDD approaches. Nodes in this diagram correspond to concrete screens. Transitions between nodes must be associated to an event on an action widget (e.g. a button) in the originating node (web page), and leading, at execution time, to the destination node (web

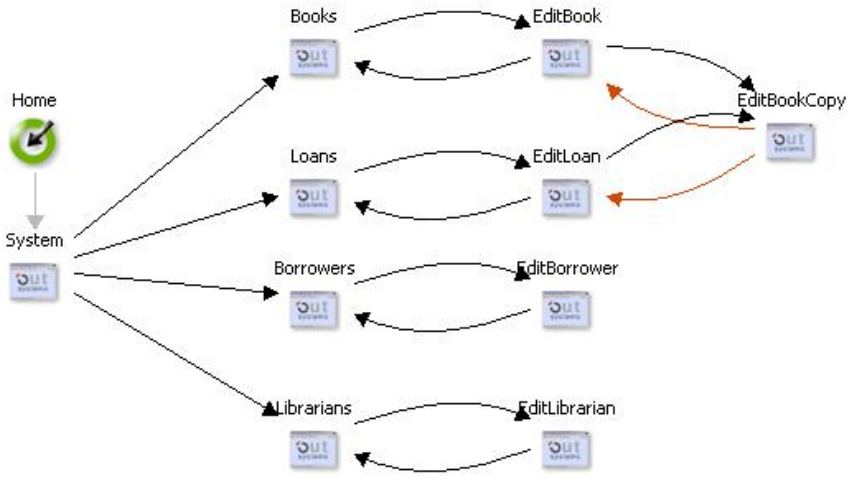

Figure 3.17: Screen flow in outSystems' Service Studio.

page). In the User Interaction Flow diagrams it is also possible to easily grant or revoke the access rights of multiple user profiles to each of the pages and transitions [Out].

Services may also be attached to action widgets. OutSystems provides a set of EntityActions and integrates a service editor that enables the definition of services through the use of a visual diagrammatic language.

3.5 Analysis and discussion of the surveyed approaches

3.5.1 Introduction

For comparing the surveyed approaches, we will consider the following questions:

1. The approach generates a fully functional application/prototype?
2. Only a prototype of the UI is generated, and no core functionality?
3. The generation process is fully or semi-automatic?
4. The generated artifacts are directly executable/compilable, or do they need a simulation environment?
5. The approach follows a model-driven development paradigm?

6. Which models are used by each approach for modeling an interactive application?

7. Which models have to be constructed by hand in each approach?

8. Which models are obtained automatically from the others?

9. Which features are present in the generated UI, in the different approaches?

For trying to answer each of these questions, a three-level detailed analysis was performed. Firstly, the demanded input for each approach is analysed, separating model-driven approaches from non-model-driven approaches. The intention is to gauge the effort demanded by each approach for UI generation, and the results that may be obtained by each approach.

Secondly, making a clear option by the model-driven approaches (remember the MDD advantages on section 2.4.4), the model architectures of the model-driven approaches are compared, in order to understand how each approach models the following aspects of an interactive system:

- **system structure** - The informational structure of the system. The system structure may be approached at two different abstraction levels. At a more abstract level, the domain model specifies the persistent domain entities, its attributes, relations (inheritance, aggregations, compositions, or simple associations), its invariant constraints, and sometimes also its operations, and respective signatures, preconditions, and post-conditions. At a less abstract level, the application (structural) model identifies not only domain entities (classes), but also all the analysis classes that are needed for accomplishing the system modeling (e.g. boundary classes) [Pre05].

- **system behavior** - How the system behaves, how it responds to stimuli, and how it enables system objects to live, collaborate with each other, and die in the scope of their life cycles. For specifying the behavior model, several kinds of representations may be used. For instance, valid object lives may be defined using a state-transition diagram (also known as a statechart or a state-machine diagram); how objects collaborate, may be specified using a sequence diagram or a collaboration (or communication) diagram [Pre05, SV05], or an action language that enables the specification of the behavior of the classes' operations. Other possible notations are, for instance, activity diagrams.

- **system use cases** - What functions does the system provide to its users. A use case model specifies the operations that are provided to users according to its roles in the system. Use-case diagrams, and some sort of use case detail specification (e.g.: activity diagram, sequence diagram, interaction overview diagram) may be used for modeling use cases and use

case scenarios. Sometimes, use cases, seen as goals that the user wants to accomplish on the system, may be detailed by using a task hierarchy (e.g.: CTT - ConcurTaskTrees) [Pre05, SV05, DFAB98].

- **system user interface** - How the system is presented to its users. What "screens" does the system show when interacting with a user in the context of a use case, and how the user may navigate through those "screens" [Pre05, DFAB98].

Finally, in an analysis of a third level of detail, a fine grained comparison is made between the model-driven approaches.

3.5.2 Features of the surveyed approaches or of the respective generated UI

In this section the features of the surveyed approaches, or the generated UI and other artifacts of the approaches, are analysed together with the demanded input of each of the approaches. Tables 3.3 and 3.4 summarize the comparison features and the demanded input of the generated UI for, respectivelly, the surveyed MDD approaches and the surveyed non-MDD approaches.

Elkoutbi *et al.* and Martinez *et al.* approaches are able to produce a UI from the structural, use case and UI behavioral models, but demand the attachment of UI related information (input/output fields and/or widgets) to the use case detail specification, respectively collaboration diagrams and message sequence charts. The generated output is only able to simulate the specified use cases through the generated UI, with no business-level application behavior.

Forbrig *et al.* approach and Wisdom are not automatic. Forbrig *et al.* base their approach on the manual selection of patterns, from a repository, that drives the model construction and transformation towards a final application. In Wisdom, the Winsketch tool helps building and validating Wisdom models, and supports the tracing of model elements through the different process phases. Despite this, no code generation is possible.

Only XIS, the OO-Method 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.

XIS allows two approaches to interactive systems generation (refer to section 3.3.1). In the dummy approach, a domain model, an actors model, a set of interaction spaces specification and a navigation space model must be fully specified.

The XIS smart approach tries to simplify the construction of the user interfaces view (interaction spaces definition and navigation space model), by demanding the construction of two other models, a business entities model and a use case model. This approach to the UI model derivation is simpler than the full construction of this, but it comes with the cost of the inflexibility of the generated UI.

	XIS approach	OO-Method	Elkoutbi et al.	Martinez et al.	Forbrig et al.	ZOOM	Wisdom
Summary of demanded input	*Dummy approach*: Domain model, actors model, interaction spaces and navigation space *Smart approach*: Domain model, actors model, business entities model, use case model	Object model, formal constraints, state-transition diagrams, presentation model (optional)	Use case model, structural model, collaboration models	Use case model, structural model, message sequence charts	Business-object model, task model, user model, device model - constructed by manual selection of patterns	Class model, finite state machine model, UI model	User role model, use case model, domain model, architectural model, navigation map, dialog model, presentation model
Generates complete application/ prototype	√	√	—	—	—	√	—
Generates only a UI prototype	—	—	√	√	√	—	—
Generates code for user defined operations	—	√	—	—	—	√	—
Existing tools support	ProjectIT-Studio	Olivanova tool set	proof of concept tool	proof of concept tool	proof of concept tool	ZOOM tool set	WinSketch (modeling for documenting tool)

Table 3.3: Features of the surveyed MDD approaches or of the respective generated UI.

The XIS smart approach demands an actors model, a use case model, a domain model and a business entities model. The latter serves the purpose of declaring Lookup and Master/Detail patterns between domain entities [Sil03, dSSSM07]. Indeed, the modeler needs to associate domain entities within business entities in order to be able to provide a lookup or master/detail pattern to the user interface needed for the interaction inside the context of a use case. The model to model transformation depicted in figure 3.6 for the smart approach is not yet available in the ProjectIT-studio, so all the XIS models must be built by hand.

It is not possible, in XIS, to specify complex behavior - only CRUD (Create, Retrieve, Update, Delete) operations may be used attaching them to Business Entities and to the connection between the use cases and business entities. Also, the need to attach a stereotype to every model element, in XIS, makes the models

	Editing model of interaction	Generic UI Programming	Generate UI from XML-schemas	AOM	Nguyen and Chun's approach	outSystems
Summary of demanded input	Serializable objects	Class definitions and generic parametric functions	XML-schema files	Metamodel, model conforming to the metamodel, and application/framework for interpreting the metamodel data.	Use case model, use case specification and sequence diagrams	Graphic construction of screen flow diagram, Web pages, entities and user actions.
Generates complete application/ prototype	✓	—	—	—	—	✓
Generates only a UI prototype	—	✓	✓	—	✓	—
Generates code for user defined operations	—	—	—	—	—	generates EntityActions and allows the graphical definition of user defined operations (userActions)
Existing tools support	demonstration tool	demonstration tool	demonstration tool	AOM is an architectural pattern and a set of prescribed design patterns	demonstration tool	outSystems Service Studio (commercial tool available)

Table 3.4: Features of the surveyed non-MDD approaches or of the respective generated UI.

hard to read and build.

In the OO-Method, also the CRUD operations are pre-defined. Nevertheless, the OO-Method allows user defined operations (services and transactions) by specifying them using OASIS (refer to section 3.3.3). Also, for not demanding the knowledge of OASIS, the OO-Method has a solution that comes with the price of flexibility: it is possible to specify how each service changes the object state depending on the arguments of the involved service and the current object state, by categorizing each attribute, and introducing additional relevant information. Possible attribute categories are push-pop, state-independent, and discrete-domain valued attributes (refer to section 3.3.3).

The OO-Method permits, as well, the specification of allowed states and state

transitions within a class. Each state transition may have attached a control (guard) or triggering condition. It is also possible to define transactions involving services of different classes.

Another OO-Method's feature is the allowance of derived attributes, by assigning a calculation formula to the attributes.

The ZOOM approach models a system using the ZOOM language, which is a formal object oriented extension of Z. Additionally, it allows the building of a graphical model, which is then translated to ZOOM. The models that are demanded by the approach, in order to automatically generate an executable application, are: a class model, which models the structure of the system and contains all the classes of the application; a finite-state machine model that models the system behavior and is the central communication mechanism to connect the structural model to the UI model; and, a UI model, which models the UI screens by using predefined components that are organized according to a user defined layout.

In what respects to the non-MDD UI generation approaches, six different approaches have been surveyed (refer to section 3.4).

In the editing model of interaction [DS90], the approach developed an editor for objects serialized in some notation, which implement an interface for persistence (load and save). It is then a simple object editor.

The generative UI programming approach [DLW05] consists in the definition of generic UI classes that may be instantiated with any class, generating UI specific classes for the parameter class. The approach may be used in the development of real applications, but it is a programming based approach, and requires the full manual coding of the business logic classes and of the classes for the mapping layer between the UI and the business logic classes.

The generation of UI forms from XML-schema files [LLK04] is a purely structural UI generation. It is based on the structure of the XML-schemas and has no application code to run, it must be coded by hand together with the mapping code between UI and the business logic classes.

Adaptive Object Model (AOM) [YJ02] is an architectural pattern and, an application that applies it must be fully developed by some means.

Nguyen and Chun's approach [NC06] prescribes an interactive modeling methodology, in which use cases are detailed using UML sequence diagrams that may be executed, in a simulation environment, in order to validate the model while it is being constructed.

Finally, the outSystems' agile approach [Out] is the one that gets closer to the MDD approaches. It allows the development of web applications both for Java based and for Microsoft.Net based Web servers. It enables the specification of a screen flow diagram, resembling the interaction navigation model found in some MDD approaches, which relates concrete web pages. It also enables the definition of entities that are used to generate database tables and may be used as constructs in the development approach. When defining entities, the tool automatically

generates what it calls entity actions, that are operations that may be used by the developer and provide the CRUD functionality for the defined entities.

Outsystems enables, then, a higher abstraction level than most commercial development tools, but is not a model-driven development tool. We could say that, using the outSystems service studio, we are constructing a platform specific model, either for Java or dot Net, but always to a relational database environment and to a web server.

3.5.3 Model architecture of each MDD surveyed approach

The models used by each of the surveyed MDD approaches are summarized in table 3.5. An *Output* mark means that the model is generated automatically by the approach. If a model is marked *In/out* it means that it may be furnished as input to the approach, or it may be automatically obtained from some of the other models. If no mark exists, it means that the model must be constructed by hand in the respective approach.

From what has been previously stated, one can see that all the MDD approaches surveyed earlier in this chapter, and referred to in table 3.5, use a class-based structural representation. The XIS, Wisdom and OO-Method approaches start with the structural modeling of domain entities. The Wisdom approach prescribes a manual process and so, an analysis and a design model for structural system aspects are also foreseen. The other four approaches demand the full specification of structural elements, not only domain entities, but also UI and control classes.

In what respects to the behavioral model of the core system, the XIS approach allows the definition of CRUD behavior over business entities (BEs), manipulatable units of structure defined over domain classes. When linking BEs to use cases, XIS permits the declaration of what operations are allowed over one BE in the context of a given use case, by declaring in the link a subset of the operations thought for the BE. This way, XIS permits a very expedite way of defining behavior and attaching it to use cases, but limits the kinds of behavior that a use case may make use of. It is not possible in XIS to specify the semantics of classes' operations. In the OO-Method and the ZOOM approach, the core behavior is defined using finite state machines (or state transition diagrams). In the OO-Method, one state transition diagram per class plus one (non-UML) object interaction diagram for the system allow the definition of complex behavior, although it might be needed the knowledge of OASIS language for specifying the semantics associated to a change of state. The knowledge of OASIS may be worked around by categorizing each attribute as explained in section 3.3.3. ZOOM has a similar approach, it allows the definition of behavior using the ZOOM object-oriented extension to the Z formal language, but the way it has to ease the knowledge acquisition process of the ZOOM language, by the modeler, is by having shaped the language in a Java-like style.

	XIS approach	OO-Method	Elkoutbi et al.	Martinez et al.	Forbrig et al.	ZOOM	Wisdom
Structural Model	Domain model + Business Entities (BE) Model	Object Model + Formal Constraints	Application Model with domain and UI classes	Application Model with domain and UI classes	Business objects Model with domain and UI classes	Structural model + Formal constraints	Domain or Business Model + Design model
Behavioral Model	Association of BE to use cases, specifying allowed operations	1 STD per class + 1 Object Interaction Diagram	—	—	—	Finite state machines (state-transition diagrams)	Analysis model + interaction model
Use cases / Task Model	Actors model + Use case model + Association of use cases to business entities	User roles represented as classes in the object model.	Use case model + Collaboration diagrams	Use case model + Message sequence charts (MSC)	Task model + User model	—	User role model + Use case model + Activity diagrams or CTT
User Interface Model	*In/Out*: Interaction Spaces Model + *In/Out*: Navigation Model + *Output*: concrete UI	Presentation Model	UI state-transition diagrams (STDs) + *Output*: concrete UI for simulation env.	UI labelled MSC + *Output*: STDs + *Output*: concrete UI for simulation env.	Presentation Model + Layout Model + Dialog model + *Output*: concrete UI	Presentation Model (MDML) + UI objects behavioral model (state chart)	Presentation model (CAP) + Dialogue model (CTT)

Table 3.5: UI generation approaches model usage

Wisdom uses user-centered design for developing interactive systems. It, then, focus its analysis on user actions, defining the core behavior of the system by an interaction model (task tree diagram) that maps to an analysis architectural model (Wisdom diagram) that categorizes classes as Task, Control or View. In Wisdom it is not possible to define the behavior of operations or state transitions.

In what concerns to the system use case model, almost all approaches allow a use case driven methodology, by defining a use case model that may already enumerate the system actors or these may be defined using a separate actors (or user roles) model. This is the case of the XIS approach, the Wisdom approach, and of the approaches from Elkoutbi *et al.* and Martínez *et al.* The OO-Method represents the actors as classes in the structural diagram, not providing a use case driven approach. Forbrig *et al.* approach also doesn't provide a use case driven methodology, but allows the definition of a user model, which is a kind of dialogue model, that specifies how a user reacts to stimuli when trying to fulfil

goals modeled in the task model. We could see the task model as a kind of enhanced use case model.

The ZOOM approach has no way of specifying a use cases view.

Finally, concerning user interface, depending on the approach taken with XIS (*dummy* or *smart* approach), it may or not be necessary to develop a UI model from scratch, as we have seen before, but a UI model is always needed. The OO Method is able to generate a user interface from the structural model, but if "less structural" UIs are needed it allows the definition of a presentation model. The approaches from Elkoutbi *et al.* and Martínez *et al.* generate a concrete UI for simulation purposes, from non-UI models, although they need to be enriched with UI information. Wisdom, the ZOOM approach and the approach from Forbrig *et al.* all need the specification of a UI or presentation model, for being able to produce a concrete UI.

3.5.4 Fine grained comparison of surveyed MDD approaches

It has been written before (refer to section 1.1) that it is a common software engineering practice to start modeling a system by building a domain model, that is a structural model composed only of domain classes, and a functional or use case model, composed of actors (or user roles) and use cases. The integration between these two system sub-models is somewhat forgotten in practice, or it is done by means of natural language text. In more complex projects and more organized teams, this integration may be accomplished by building, for instance, a sequence model. We believe that it is possible to accomplish this integration by enriching these two models using OCL.

In order to see which model elements are used in the main surveyed approaches to build the domain model and the use case model, a set of model elements and explored features were compared. Tables 3.6 and 3.7 explain the meaning of the compared model elements and features, for the domain and use case models. Table 3.8 summarizes a fine-grained comparison of the domain model elements used by the surveyed MDD approaches, and table 3.9 shows the summary of a fine-grained comparison of the use case model elements used by each approach.

From tables 3.8 and 3.9, and from the above discussion, one can identify the main flaws of each of the surveyed approaches, which are summarized below.

XIS has no way of specifying the signature and the semantics of operations. The use of OCL could help in enriching the semantics of the domain model. XIS views definition mechanism (BusinessEntities) is very simple and expedite, but turns to be inflexible when wanting to include, for instance, calculated fields in the UI.

XIS smart approach will, when available, make possible to obtain a default user interface model based on the domain, business entities, actors and use case models. The default UI will be guided by the use cases. Although very simple, the XIS way of specifying use case behavior, by attaching a use case to a business

Submodel	Element or feature	Description
Domain Model	Classes	Concepts or entities of the problem domain. Classes are defined in terms of attributes, operations and relationships.
	Attributes	Properties of the concept or entity modeled by a class.
	Relationships	Associations between classes (e.g.: aggregation, composition, simple association), or relationships of type "is a" between two classes, i.e. generalization relationships.
	Class invariants	Conditions that instances of the corresponding class must obey.
	Attributes default values	Attributes may have defined default-values.
	Derived attributes	Attributes may have a value that is calculated dynamically according to a given formula.
	Mandatory attributes	Attributes to which a value must be provided, when creating an object.
	Read-only attributes	Attributes to which a value may only be provided when creating an object. After creation the value of the attribute may not be modified.
	Operations or methods	Services provided by the instances of a class. The syntax of an operation is defined by its name, and the type and order of its parameters and of the result yielded by the operation.
	Operations' executable semantics	The semantics of an operation may be defined in an implicit way, by specifying its pre-/post-conditions, or in an explicit way by specifying its denotational semantics or, less abstractly, its sequence of actions. Typically, only the latter may directly or indirectly execute an operation's semantics.
	Enumerations	Enumeration classes are enumerated types or static lists of discrete values.
	Views	Classes whose attributes are taken from other classes (Views). Views / Derived classes usually model documents of the domain that mix attributes from more than one entity or domain class.
	Triggers	A trigger is a behavior that is automatically executed whenever a given operation is invoked and/or a given condition is true. If triggered by an operation invocation, a trigger may occur before, after or instead of the operation that triggers it. If triggered by the occurrence of a given state condition, the trigger is automatically invoked whenever the condition is true.

Table 3.6: Main domain model elements

entity and identifying allowed operations, limits the kind of allowed operations to CRUD, even when guiding the UI generation by the use cases. One way to turn around this limitation could be the exploration of use case relations (e.g.: inclusion and extension), or the use of constructs typical of task hierarchies specifications, like sequencing or alternative, to detail use cases and allowable use case navigation by an actor.

The OO-Method and the ZOOM approach, are richer in what respects to the domain model, but they have no views specification mechanism. ZOOM always requests the construction of a UI model, having no way of deriving one from the other models of the system. When the UI structure is not based on the domain

Submodel	Element or feature	Description
Use case model	Actors / Roles	Actors / roles for using the system.
	Use cases	Use cases are intended as system packaged functionality.
	Use case relations	Relations between use cases (e.g.: inclusion, extension, generalization).
	Mapping use cases to domain classes	Each use case must be mapped to domain classes by some means. The use of UML interaction diagrams is the typical means. Other ways may involve the detailed specification of use case behavior, either by specifying pre-/post-conditions in natural language text or in a formal notation, or by subdividing use cases into sequencing or alternative atomic actions that are mapped to domain classes.
Task Model	Task	The approach allows the definition of tasks, intended as activities that the user must perform on the system, in order to achieve a goal, or that the system may perform in response to a stimulus. Tasks can be defined at different levels of abstraction allowing the definition of sub-tasks.
	Action	A concrete (final) task that can be executed. It may be a user action or a system action.
	Sequencing	The temporal order that sub-tasks and actions must respect for carrying out the related high-level tasks.
	Other task operators	Besides sequencing actions, other task operators may be used in task models (e.g.: alternative tasks, task enabling, task disabling) [Pat03, DFAB98].

Table 3.7: Main model elements of use case / task model

classes structure, the OO-Method also demands the construction of a UI model. So, there is no way of deriving a default UI model guided by use cases, in the OO-method.

The approaches from Elkoutbi *et al.* and Martinez *et al.* derive a UI model guided by use cases, from other models. Nevertheless, their approaches are very laborious, demanding the manual labelling of use case scenario definition models, respectively collaboration models and message sequence charts, or state transition models, with UI related information. The resulting UI may only serve to animate the specified use cases in a simulation environment, with no business level behavior.

Forbrig *et al.* approach is manual, although it may be computer assisted, easing the process of choosing applicable UI patterns. The only step that may be automatized is the final step of producing a concrete UI from the constructed pattern-based UI model.

Finally, the Wisdom approach, is a process and a method for developing interactive applications. It doesn't foresee the specification of operations' semantics. It is a model-based approach, but the tools available (e.g.: WinSketch) don't provide a model-driven environment. The data in tables 3.8 and 3.9, for Wisdom, were based in WinSketch. Also in Wisdom, the UI model must be fully

		XIS approach	OO-Method	Elkoutbi et al.	Martinez et al.	Forbrig et al.	ZOOM	Wisdom
Base classes	Classes	√	√	√	√	√	√	√
	Attributes	√	√	√	√	√	√	√
	Relationships	√	√	√	√	√	√	√
	Class invariants	—	√	—	—	—	√	—
	attributes default values	√	√	—	—	—	√	√
	derived attributes	—	√	—	—	—	—	—
	mandatory attributes	—	√	—	—	—	—	—
	user defined operations' syntax	limited (only operation name)	√	—	—	—	√	√
	user defined operations' semantics	—	√	—	—	—	√	—
Lists of values	Enumerated classes	√	—	—	—	—	√	√
Views	Views / derived classes	√ (business entities)	—	—	—	—	—	—
	Mapping to base classes	√	—	—	—	—	—	—
Triggers or other forms of modifying CRUD		—	—	—	—	—	—	—
Operations triggered by state conditions		—	√	—	—	—	√	—

Table 3.8: UI generation MDD-approaches fine-grained comparison of the domain model elements

constructed by hand.

Table 3.10 compares and summarizes features of the surveyed MDD approaches that are able to automatically generate a UI prototype.

Only XIS, the OO-Method and ZOOM can generate a fully functional interactive prototype. Martinez and Elkoutbi approaches only generate a non-functional, although navigable, UI prototype.

All the approaches present in table 3.10 require the full specification of a UIM, although the XIS approach has plans for its future derivation from other provided models. Then, the XIS approach will also be able to generate a UI prototype

		XIS approach	OO-Method	Elkoutbi et al.	Martinez et al.	Forbrig et al.	ZOOM	Wisdom
Actors (user roles)		√	User roles represented as classes in the object model.	√	√	√	—	√
Use cases		√	—	√	√	√	—	√
Use case extension		√	—	√	√	√	—	√
Use case inclusion		√	—	√	√	√	—	√
Mapping to base classes	Target views / operations	√	—	—	—	—	—	√
	behaviour	—	—	Sequence diagrams	Collaboration diagrams	Task model	—	Activity or task diagrams(CTT)

Table 3.9: UI generation MDD-approaches fine-grained comparison of the use case model elements

from the domain, use case models and other demanded non-UI models.

Only the OO-Method is able to generate a UIM and a UI prototype from the domain model alone. And, it is also the only one that allows the definition of triggers activated by a condition.

XIS and the OO-Method assume the existence of CRUD operations in the context of domain entities. But, only ZOOM and the OO-Method are capable of generating code for user defined operations.

Finally, only the ZOOM approach takes advantage of formal constraints to generate features in the UI.

3.6 Conclusions

This chapter surveyed seven MDD approaches to interactive applications, or user interface, generation, and six other approaches to the development of user interfaces. The advantages of the MDD approaches over the others have been identified, although some of the other approaches may have, in certain conditions, more expedite, but less flexible, ways of developing interactive applications.

In the above analysis and discussion of current approaches and tools for model-driven interactive applications generation, we have shown that only XIS, and in a sense the OO-Method, are able of deriving a UI model from other system models. But the generated UI model provides only CRUD operations, and may only be based in the domain structure or, in the case of XIS, in a defined set of views (BusinessEntities) established over the domain classes.

	XIS approach	OO-Method	ZOOM	Elkoutbi et al./ Martinez et al.	Forbrig et al.
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	generates only UI state model	requires
Is able to generate a UIM/UIP from non-UI system models	√ (in smart approach)	(only from domain model)	—	√ (non funtional 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 3.10: Feature comparison between the current approaches.

Only XIS is able to derive a default use case driven UI model, but, as we have seen, it is based on a semantically poor domain model and is limited to CRUD operations specified over the business entities.

From this chapter's state of art survey and discussion, the main flaws of existing approaches to UI automatic code generation have been identified, and are summarized bellow:

- In general, current approaches demand too much effort, from the modeler, in order to build the system models demanded as input by the approaches. They don't allow a gradual approach to system modeling 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 [PMI04,

Mol04, PIP+97], 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 [dSSSM07], that enables the generation of a *default* user interface from the core system models, but the generated UI is very limited in what concerns the core system behavior.

- 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. Nguyen and Chun's approach to MDD [NC06] allows the interactive modeling and animation of use case driven UML models, but it is based on the direct animation of the models in a simulation environment, not in the animation through a derived UI, so it may not be used by users or other stakeholders than the modeling team themselves.

- 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 [JSL+05, JSQ+07], and partially the OO-Method [PMI04, Mol04, PIP+97].

- Existing approaches don't take advantage of the use of constructs typically found in task models (e.g.: sequencing, alternative) for detailing use cases.

- Existing approaches don't allow the definition of the semantic of operations at class level. Again, the exception is the ZOOM approach [JSL+05, JSQ+07], and partially the OO-Method [PMI04, Mol04, PIP+97].

- With the partial exception of the OO-Method [PMI04, Mol04, PIP+97], existing approaches don't allow the definition of triggers, that are operations triggered by the invocation of other operation or by the holding of a given condition. Triggers activated by an operation's invocation are a way of modifying or adding behavior 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.

The approach to UI model generation defended in this PhD research work addresses the issues referred to in tables 3.8 and 3.9, as presented in the next chapters.

Chapter 4

Proposed Process and Metamodels

This chapter presents a model-driven software development process, that may be instantiated from any iterative incremental process, which is based on the automatic generation of the final application from platform independent models. The chapter also presents the UML-aligned metamodels used for developing the system model, and the MOF-based metamodel for creating UI models.

4.1 Introduction

The approach to model-driven UI generation and interactive applications development, proposed in this dissertation, enables the automatic generation of user interface models (UIM) and executable user interface prototypes (UIP), from early, progressively enriched non-UI system models.

The reason for obtaining an intermediate UI model is for easing the transformation process and increase its productivity, allowing to obtain a platform independent UI model that enables the use of specialized model to code (M2C) generators to different architectures or different target platforms. This is completely aligned with MDD.

In contrast with other approaches (refer to chapter 3), our approach doesn't require the modeler to build a UIM (see Fig. 4.1). In fact, a UIM may be automatically generated from the domain model alone, or from the integrated domain and use case models, and then be subject to a M2C process in order to obtain the executable application or prototype final code.

The approach invites the modeler to start by building a domain model (DM), generate a UIM through process DM2UIM (see Fig. 4.1), and finally generate and execute the prototype in order to validate both the DM and the UIM. This may occur during several iterations, during which the DM is refined and enriched, until it is not sufficient. Indeed, the DM will not be sufficient if the modeler needs to specify different functionality for different actors, or if he needs to specify

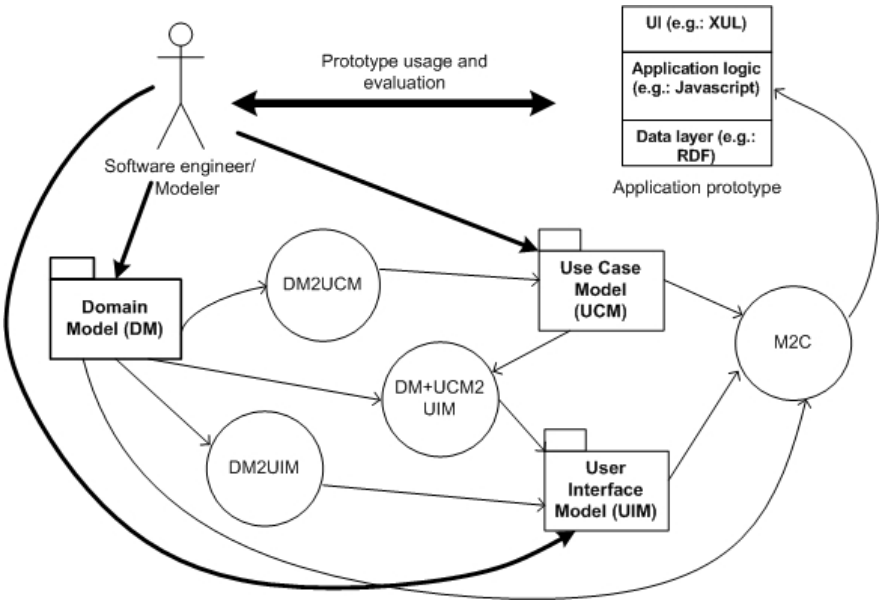

Figure 4.1: General approach to UI generation.

functionality that cannot be fully derived just from the structural system specification, that is what is defined in the DM. When this is the case, the modeler will need to build a use case model (UCM) integrated with the DM, and for that he may use model-transformation tool DM2UCM that will generate a default UCM from which the modeler may start working. This way, the modeler is able to start building a UCM, not from the scratch, but from the full functionality that may be derived from the previously built DM. This is, of course, an optional step.

Then, for several iterations, the modeler may refine and enrich both the DM and the UCM, in a fully integrated manner. In each iteration, the model-transformation process DM+UCM2UIM is used to generate a UIM, from which a prototype is automatically generated and may be, then, assessed in order to validate the system models (DM, UCM and UIM).

Finally, the modeler will need to refine the UIM directly and, after that, only by modifying the generated code, by hand, will he be able to refine the generated application, assuming that he has no access to modify the M2C process.

This approach is embodied in a process defined in the next section. Of course, the defined DM, UCM and UIM metamodels don't confine the modeler to follow the described approach, but if he wants to be able to automatically generate both a UIM and an executable prototype and application, he will need to follow it.

The following sections describe the referred metamodels.

Both the proposed process and the metamodels formalization, presented in this chapter, have been addressed in [dCF10].

A proof of concept tool has been developed for fully automating the DM2UIM,

DM+UCM2UIM and M2C processes (see chapter 6).

4.2 Process for model-driven development of interactive form-based applications

The approach proposed in this dissertation comprises an iterative development process that enables the automatic generation of UI models and the development of form-based data intensive applications, from early progressively enriched platform independent models, by following a MDD paradigm. After the modeling activity and the UIM generation step, in each iteration, the approach permits the generation of an executable user interface prototype (UIP), which enables the complete model validation by other stakeholders besides the modeler himself.

The process, illustrated in Fig. 4.2, comprises three phases:

1. Requirements Modeling and Validation;
2. UI Design and Validation;
3. Implementation and Validation.

In the first process iterations, phase 1 activity, that is Requirements Modeling and Validation, is concerned with eliciting and gathering requirements and modeling them in a domain model and a use case model, and validating the constructed models with the stakeholders, through an automatically generated application prototype. In the first iteration of this phase, the modeler has two options:

- Modeling the functional requirements as use cases by developing an abstract and incomplete use case model. Further iterations will allow building a domain model and refining the use case model by fully integrating it with the meantime developed domain model.
- Modeling requirements that are concerned with the information structure, by developing a first domain model, and generating a default use case model from the DM. Further iterations will enable the refinement of both the UCM and the DM.

In subsequent iterations the UCM and the DM are kept tightly related.

From what has been mentioned before, the recommended procedure for phase 1 is to start building a simple domain model, 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 a model to model transformation process from DM to UIM, DM2UIM, followed by a model to code transformation

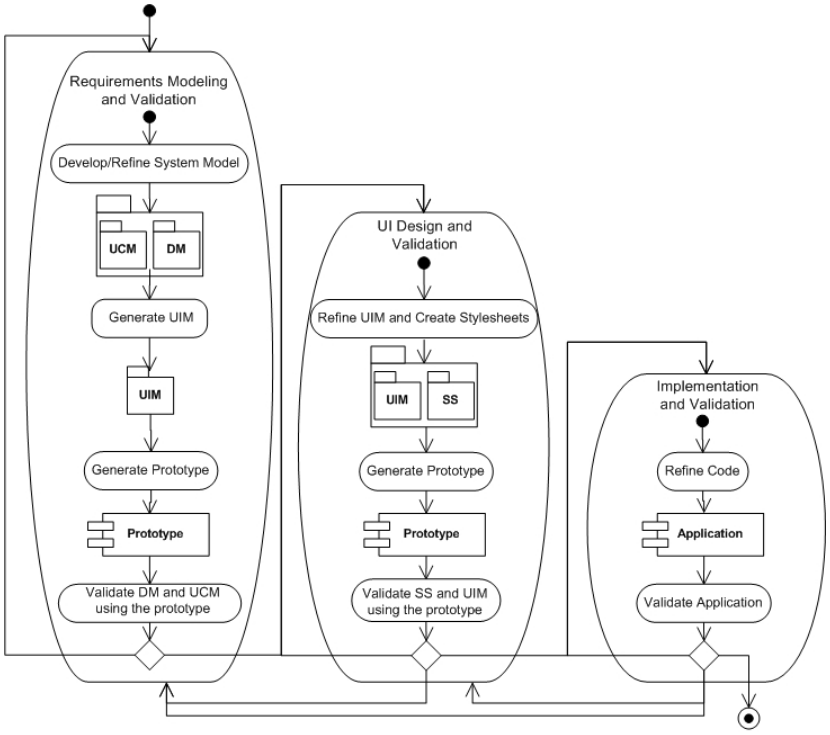

Figure 4.2: Proposed interactive applications development process.

process, M2C) - see Figs. 4.1 and 4.3 - supporting only the basic CRUD operations and navigation along the associations defined.

In subsequent iterations, the DM can be refined and extended with additional features (to be explained in more detail in section 4.4) that allow the generation of richer user interfaces: OCL constraints, default values, derived attributes, derived entities, user-defined operations, and triggers. From this enriched domain model, 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 entities (Views) allow the generation of UI forms with a more flexible data structure.

Simultaneously, the modeler may develop a use case model (UCM), integrated with the DM. This UCM will enable the separation of functionality by actor, and its customization (e.g.: hiding functionality from some actors). Corresponding UI model and UI prototype are then automatically generated from both the DM and UCM (DM+UCM2UIM and M2C processes in Fig. 4.1). As will be explained when presenting the metamodel, there is a full integration between the UCM and the DM, as use case specifications are established over the structural domain model. A first UCM may be automatically obtained through DM2UCM transformation process (see fig. 4.1).

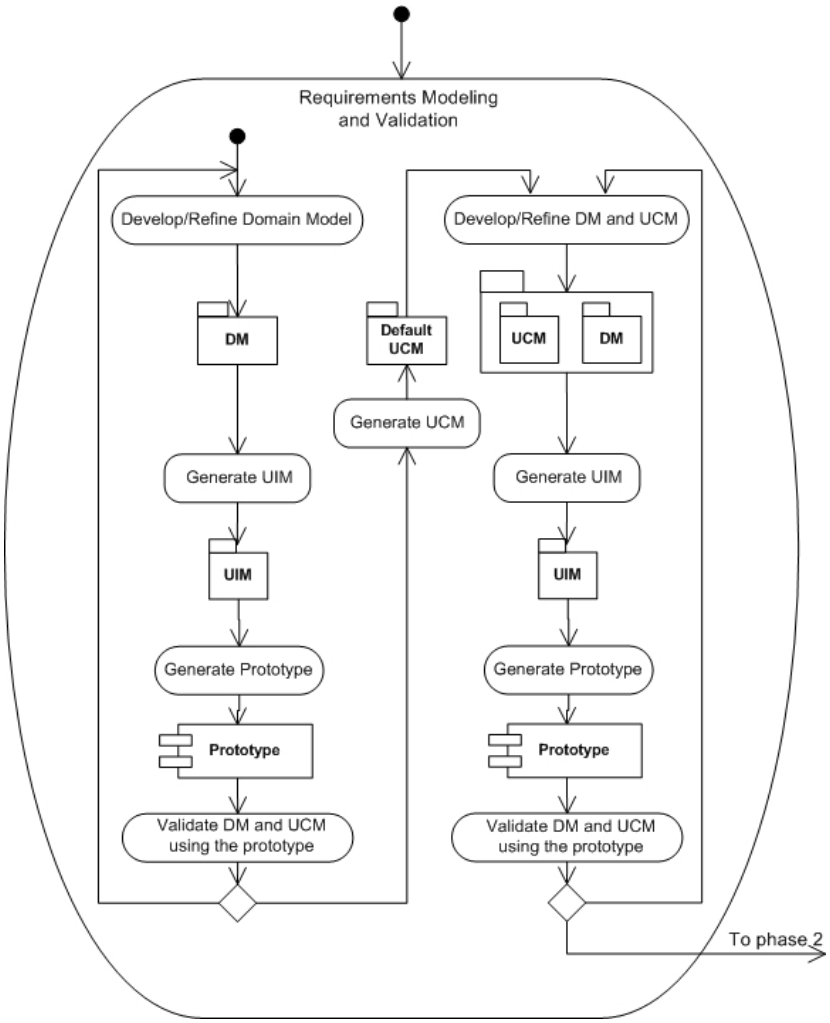

Figure 4.3: Detail of development process phase 1, Requirements Modeling and Validation, as prescribed for fully taking profit of automatic generation processes.

Phase 2 activity, UI Design and Validation, corresponds to the development of a user interface model and of concrete look & feel elements (e.g. stylesheets). It starts with the automatic generation of a UI model, wich can be later manually refined and for which a set of style classes may be defined.

Implementation and Validation is the third and final process phase, and corresponds to the later iterations of the process. Although code is obtained in all iterations of all phases, as a running prototype, the Implementation and Validation phase is where the modeler is addressing it as the code for the final application.

Each time a prototype or application is generated it is validated with the stakeholders with the purpose of, in phase 1, validating the domain and use case

models, in phase 2, validating the UI model and style sheets, and in phase 3, validating/testing the final application.

On each iteration of phases 2 and 3, 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, 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.

The developer may refine the generated code, but the recommended practice is to refine the DM and UCM models and act in the UI look & feel through the development of style sheets. When this is not possible, it may also exist a problem with the model transformation processes or with the code generation process, used in the described activities.

This software development process must be seen as being framed within a more broader industrial process that includes, as previous activities, the ones of defining the transformation rules between the metamodels instances, and of configuring code generation tools with an established architecture or target specific platforms. In this broader process, the establishment of an architecture and the selection of the target specific platforms must be addressed, in order to be able to select from the pre-configured code generation tools that must exist in the market.

This activity of establishing an architecture, and selecting final specific platforms and the appropriate code generators has only to be finished before the Implementation and Validation phase, because only there the final code is generated.

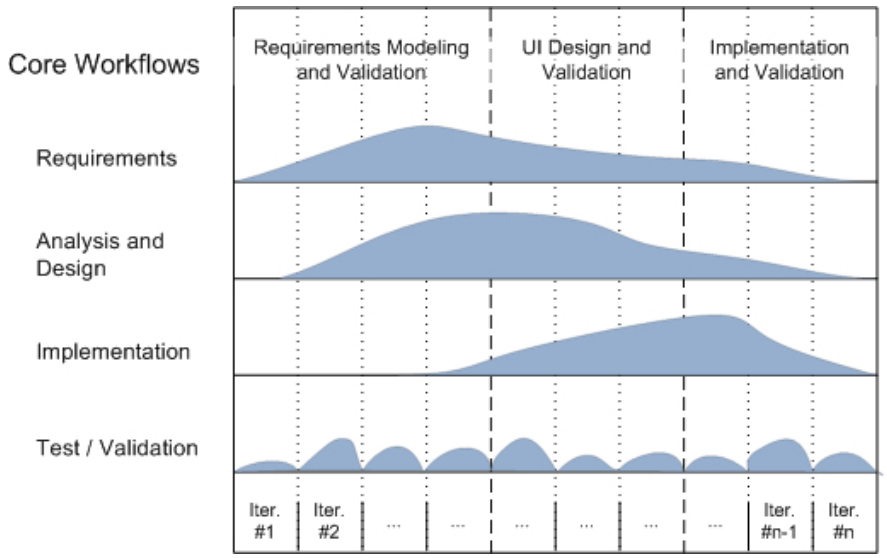

Figure 4.4: The way the core workflows take place over the process phases.

The presented development process can be related to the core software development workflows, namelly Requirements, Analysis and Design, Implementation and Test, as illustrated in figure 4.4.

4.3 Model Architecture

Each of the models referred to before (DM, UCM and UIM) is an instance of a defined metamodel. Fig. 4.5 shortly illustrates the trace relations between the UI metamodel (UIMM) and the domain model metamodel (DMM) and use case metamodel (UCMM).

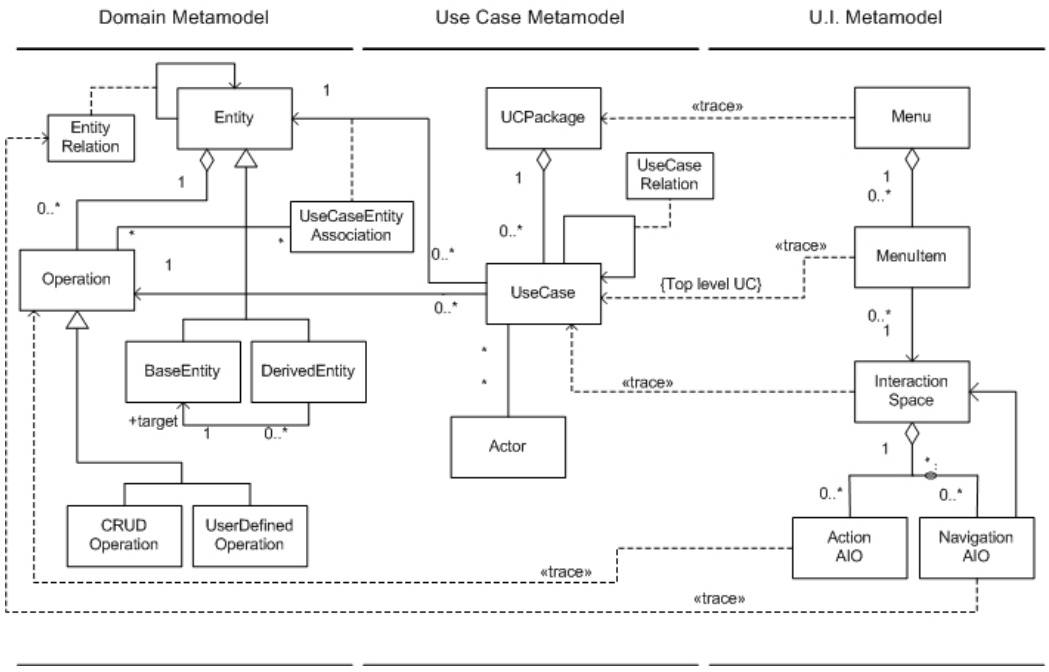

Figure 4.5: Excerpt of the conceptual metamodels and their relations.

When a user interface model is obtained from a DM and a UCM, its elements are traced back to elements in those two models, 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;
- An Interaction Space may be traced back to a use case, which is typically related to a base or derived domain Entity;

- An Action AIO may trace back to a CRUD or user-defined operation that may be related to a use case.

Automatic model transformations will be addressed in chapter 5.

The proposed model architecture allows a platform independent system modeling according to 3 views:

- a structural view, that is established by developing a domain model (DM);
- a functional view, which can be defined by constructing a use case model (UCM); and,
- a user interface view, that may be defined through a user interface model (UIM).

Figure 4.6 contextualizes the metamodel defined for this approach by dividing it in three packages corresponding to the referred model views and relating them to the UML and MOF.

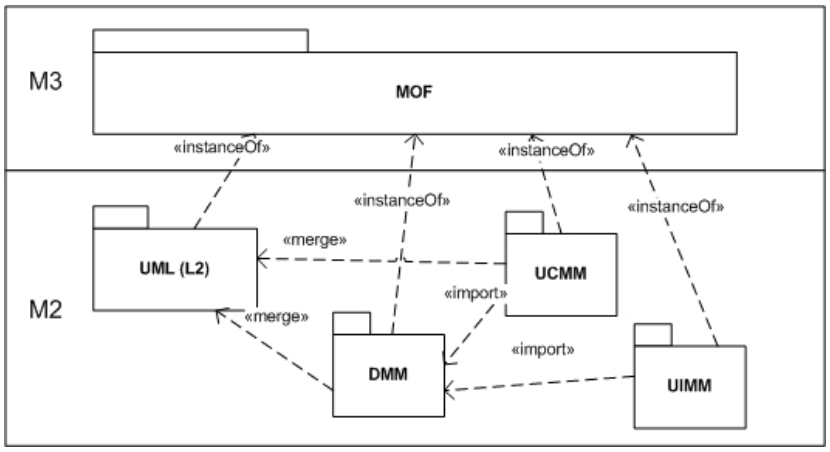

Figure 4.6: Metamodel contextualization.

MOF is the meta-metamodel that provides the concepts for defining the UML metamodel, which is stratified in language units used for defining compliance levels. Compliance level 0 (L0) has a single language unit and is formally described as the UML Infrastructure. The UML Infrastructure forms the package Core that is shared between MOF and the UML metamodel. Level 1 (L1) adds language units for use cases, interactions, structures, activities and actions, and level 2 (L2) adds language units for state machine modeling and profiles. UML has as well a level 3 (L3) compliance level that adds language units for information flows, templates and model packaging [OMG09b].

The enriched Domain Model Metamodel (DMM) is partially merged with UML L2, namely with the packages for structures, state machines and actions language units, and this way it extends UML by incrementally adding features to some of the UML L2 metamodel elements, specializing them, or adding new metamodel elements.

The Use Case Metamodel (UCMM) is also merged with UML, namely with the packages of the language unit for use cases.

The User Interface Metamodel (UIMM) is defined in conformance to the MOF meta-metamodel and imports some features from the DMM, which enable model integration.

The model architecture presented in this dissertation is, thus, aligned with the UML and the MOF.

4.4 Metamodel for Domain Models

As already mentioned, the metamodel proposed in this dissertation is built upon the UML metamodel, incrementally adding features to it through the "merge" package dependency relation (refer to section 2.6). The elements of the UML metamoldel relevant to the definition of the Domain and Use Case Metamodels have been presented in section 2.5.

Besides classes (domain entities), attributes and relationships, a 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 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 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 entities (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 metamodel package for enriched domain models is depicted in Figures 4.7 and 4.8. The reused UML elements are shaded, and among those, the ones that have been modified, either by adding or specializing features, have only the name compartment of the visual element shaded. In what follows, the modifications to UML elements are explained, followed by the description of the DM metamodel's main concepts.

4.4.1 Modified UML Elements

The DM metamodel extends some UML constructs, as can be seen in Fig. 4.7. The modified UML elements are Class (Entity), Property and Operation.

Class or Entity

It has been restricted to simple inheritance and is set to non-instantiable. Also, it has been added an attribute, isNavigationRoot, that enables the identification of an entry point for navigating in the structure. This is useful when generating a UIM from the DM alone, without the development of a UCM.

Property

A new attribute has been added to Property (isIdent) to be possible to identify entities' properties that are used by the business user as an instance identification or summarization structural feature.

Operation

The merge increment in the Operation class has turned possible to have, besides operations that are user-defined in an Action Semantics-based action language, the basic CRUD operations (Create, Retrieve, Update and Delete) that are considered to exist by default in every BaseEntity. A CRUD operation is a high level transactional operation, whith the semantics of persisting, retrieving, updating or deleting a BaseEntity instance from a platform independent instance collection base (see section 4.4.3).

4.4.2 New Domain Metamodel elements

The DM metamodel introduces new concepts that extend the UML metamodel and ease the purpose of constructing a complete and rigorous platform indepen-

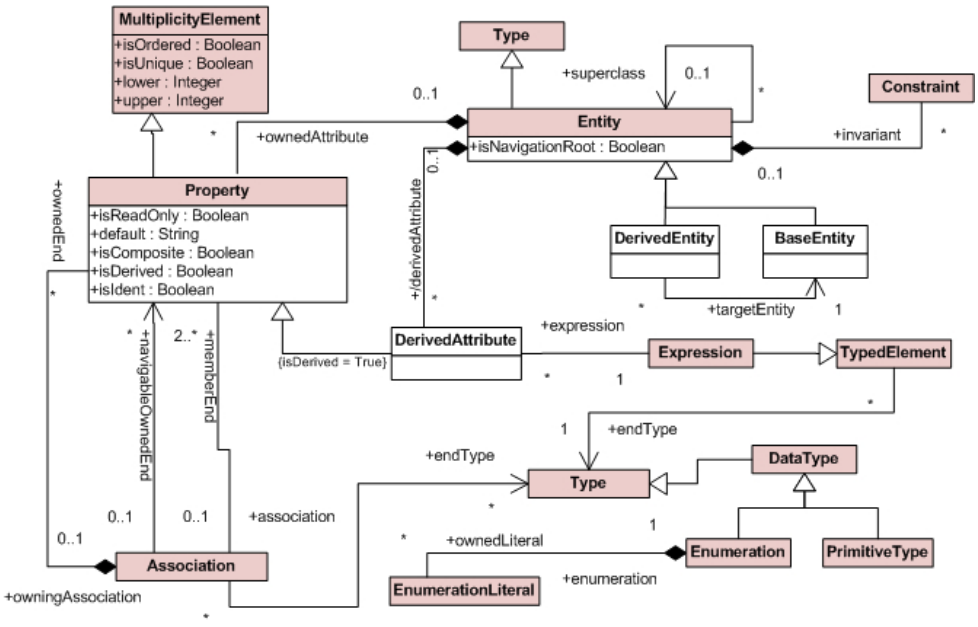

Figure 4.7: Metamodel for defining Domain Models (structural features).

dent model. The new elements are BaseEntity, DerivedEntity, DerivedAttribute, and DomainTrigger.

BaseEntity

A BaseEntity models a problem space persistable base concept in a platform independent manner. A BaseEntity specializes the modified UML Class (Entity) and inherits all its features, semantics and concrete notation.

DerivedEntity (View)

A DerivedEntity models an interesting view in the problem space (e.g. a business view). A DerivedEntity inherits all the features, semantics and concrete notation of the modified UML Class (Entity), and specializes it in order for it to be non-persistible, and that all its properties are derived attributes. DerivedEntities are non-persistent domain entities with a structure closer to the business domain, like a business document, and so closer to the UI needs. A DerivedEntity must target a BaseEntity that acts as the root for referencing derived attributes. It is treated essentially as a virtual specialization of the target BaseEntity, possibly restricted by a membership constraint and extended with derived attributes. A Derived Entity may be distinguished from BaseEntities, in a domain model, by having its name preceded by a slash in a concrete notation.

DerivedAttribute

A DerivedAttribute is an attribute that is calculated or copied from a related entity. It inherits from the UML Property class and may be used within BaseEntities and DerivedEntities. Its definition demands the specification of an Expression upon which the attribute's value is calculated. The Expression may be the declaration of a role path from the target entity to another related entity's attribute, or it may be the definition of an expression, in a subset of OCL, that may perform some calculus or aggregation. A common special case is a reference to a related attribute, using a sequence of dot separated role names ending with a property name. The concrete notation of a derived attribute has, as defined by UML, its name preceded by a slash.

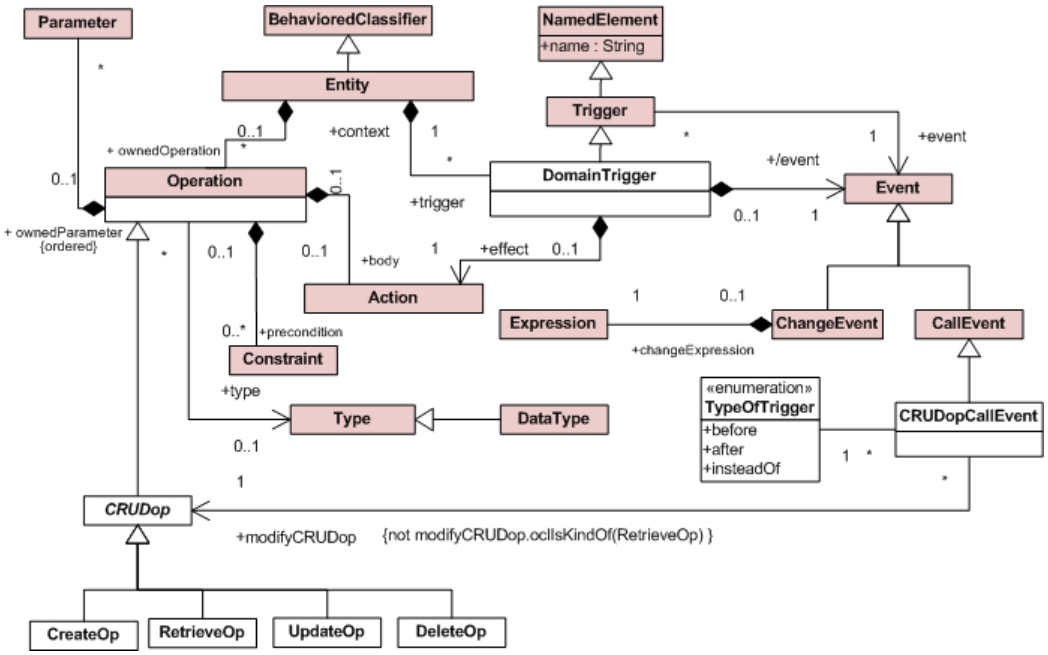

Figure 4.8: Metamodel for defining Domain Models (behavioural features).

DomainTrigger

The DM metamodel includes constructs for defining domain triggers, which are defined in the context of an Entity (see Fig. 4.8). By defining domain triggers, the modeler is able to modify the normal behavior of CRUD operations, or define generic business rules. A DomainTrigger inherits from the UML Trigger, but it has only two possible kinds of events associated, namely:

- ChangeEvent: It is the UML ChangeEvent class but with the changeExpression association end restricted to Expression type. In standard UML

Context Class	Invariant
Entity	Entity ->allInstances() ->forAll(x, y | x ≠ y implies(not x.isNavigationRoot or not y.isNavigationRoot))
DerivedEntity	self.derivedAttribute = self.ownedAttribute

Table 4.1: Domain Metamodel elements' new invariants.

it can be a ValueSpecification, that includes the possibility of defining an OpaqueExpression, which promotes the definition of platform specific expressions (e.g. Java expressions) within a PIM, which is considered to be a bad modeling practice [Fra03]. The ChangeEvent triggers a DomainTrigger when the condition defined in the changeExpression holds.

- CRUDopCallEvent. It is a specialization of UML's CallEvent, restricted to CRUD operations, and with the possibility to intercept the call to an operation before, after or instead of calling it. It provides a way of modifying the default behavior of CRUD operations. A CRUDopCallEvent triggers a DomainTrigger, before, after or instead of an identified CRUD operation call, within the context of an instance of a class, enabling the reinforcement of business rules.

By using OCL, Table 4.1 formalizes the invariants for the modified UML features and for the new DM metamodel features. Only the constraints that cannot be inferred from the presented diagrams and that are not already specified in the standard UML are shown.

4.4.3 Action-Semantics-based actions language

As mentioned in section 2.5.1, citing [OMG09b], an actions concrete language could provide high-level constructs that would combine lower-level actions. This way, in this dissertation it is considered that the actions language provides constructs for:

- Creating and persisting entity instances, and simultaneously accept attribute values that are modified in a transactional way, including association end property values that link two instances of different or same entities;

- Updating persisted entity instances by accepting attribute/value pairs, also in a transactional way, including properties linked to association ends;

- Retrieving an entity instances by searching all entity instances and applying a filtering condition;

- Deleting an entity instance;
- Declaring triggers after, before or instead of CRUD operations.

Table 4.2 shows the correspondence between the high-level constructs we are considering and the constructs provided by UML Action Semantics.

Proposed Action Language	UML Action Semantics
Entity.create($att1 = val1, ...$)	CreateObjectAction + AddStructuralFeatureValueAction for each attribute + CreateLinkAction for each Entity property that is an association end
Entity.retrieve(*condition*)	Entity.allInstances−>select(*condition*)
instance.update($att1 = val1, ...$)	RemoveStructuralFeatureValueAction/ AddStructuralFeatureValueAction for each attribute being updated
instance.delete()	DestroyObjectAction

Table 4.2: Proposed action language constructs.

4.4.4 Example

Fig. 4.9 illustrates an example of a domain model that conforms to the presented domain metamodel.

Class System is the navigation root ($isNavigationRoot = True$) for the specified entities. System aggregates base entities Book, Loan, Librarian and Borrower, and derived entity (business view) ActiveLoan.

Note that some entities have attributes stereotyped with «ident», which means that they are used by the business users as a means to distinguish between entity instances.

Entity Loan has a user defined operation, returnBook, which has one input parameter, sysDate, and yields no result. The operation body, not shown in the picture, is specified by using an Action Semantics-like language (see sections 2.5.1 and 4.4.3), as follows:

```
Context Loan::returnBook(in sysDate: Date)
body: self.update(
effectiveReturnDate = sysDate,
status = LoanStatus.Inactive
)
```

The proof-of-concept tool, reported in chapter 6, only works with abstract syntax, so the presented concrete syntax for operations body or domain triggers is only an example language.

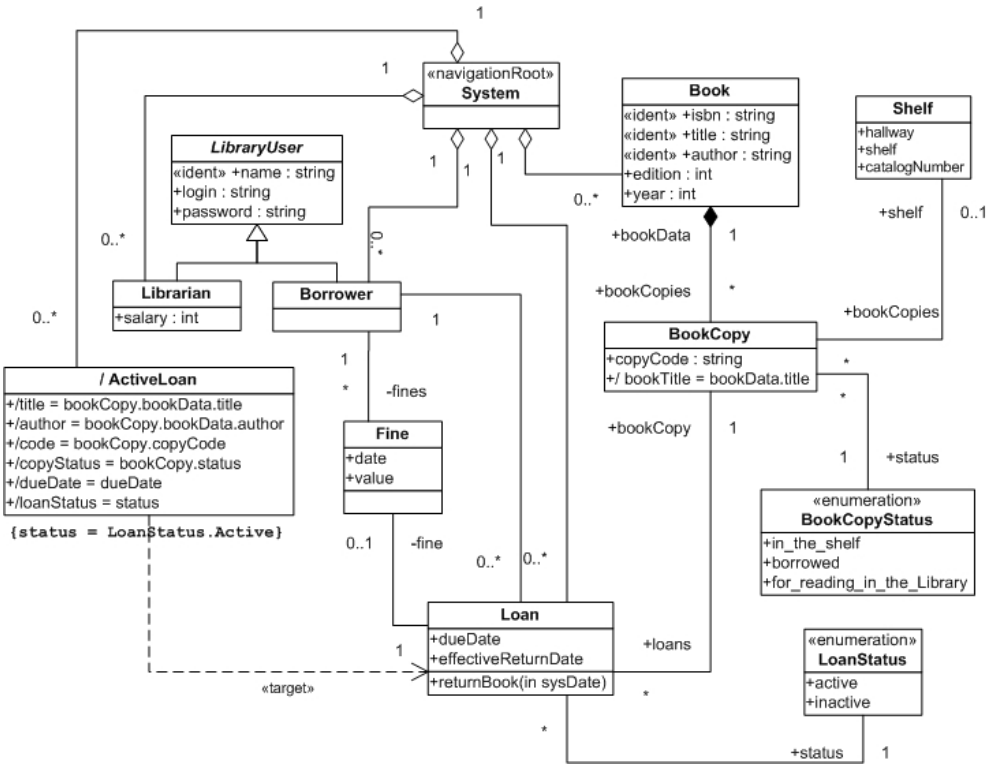

Figure 4.9: Domain model for a Library Management System (LibrarySystem).

Entity BookCopy has a derived attribute, title, defined by referencing Book's attribute title through the assotiation role bookData, from a bookcopy to its related book.

Derived entity ActiveLoan targets the base entity Loan, and filters loans selecting only the ones that have status Active. ActiveLoan has several derived attributes defined by referencing remote attributes reachable through role chains starting in the target entity, that is Loan. For instance, ActiveLoan's attribute title references the book title related to the active loan, through a role chain from Loan to Book, which refers role bookCopy, that identifies the BookCopy instance related to a given Loan, and then bookData, which refers to the Book instance related to a given BookCopy instance.

Although not shown in the figure, invariant constraints and operations preconditions could also be defined by using OCL, as for instance:

Context Book inv : self.title $\neq$ “”

or,

Context BookCopy inv Copycodes_are_unique:
BookCopy −>forall(x, y | x $\neq$ y implies x.copyCode $\neq$ y.copyCode)

Again, as the proof-of-concept tool only works with abstract syntax, OCL is only one of the concrete languages that could be used, but it seems as the most appropriate choice, as all the approach builds around UML metamodel.

Also not shown in the example are domain triggers, which could also be defined by using the operations body Action Semantics-like language (see sections 2.5.1 and 4.4.3). An example domain trigger, enforcing a business rule stating that a fine must be created every time a lent book is returned after the due date, could be defined as:

Context Loan trigger after update:
effect: if (self.effectiveReturnDate < self.dueDate)
Fine.create(
date = self.effectiveReturnDate,
value = 1,
borrower = self.borrower,
loan = self
)

4.5 Metamodel for Use Case Models

The metamodel package for use case models, shown in Figures 4.10 and 4.11, specializes and extends the UML language unit for use cases [OMG09b]. As mentioned before, a UCM can be defined in close connection with the DM, to specify and organize the CRUD, user-defined or navigational operations over Base or Derived Entities that are available for each actor. The definition of a UCM also enables the use of several features, such as task-model-like relations, that permit a fine tuning of the interaction within a use case. The data manipulated in each use case is typically determined by the domain entity (base or derived) and/or operation associated with it. Several constraints are posed on the types of use cases and use case relationships that can be defined. The UCM metamodel reused UML elements, present in Fig. 4.10 and 4.11, are shaded and, among those, the ones that have been modified, either by adding or specializing features, have only the name compartment of the visual element shaded. In what follows, the modifications to UML elements are explained, followed by the description of the UCM metamodel's main concepts.

4.5.1 Modified UML Elements

The UCM metamodel extends some UML constructs, as can be seen in Figs. 4.10 and 4.11. The modified UML elements are UseCase, Extend and Operation.

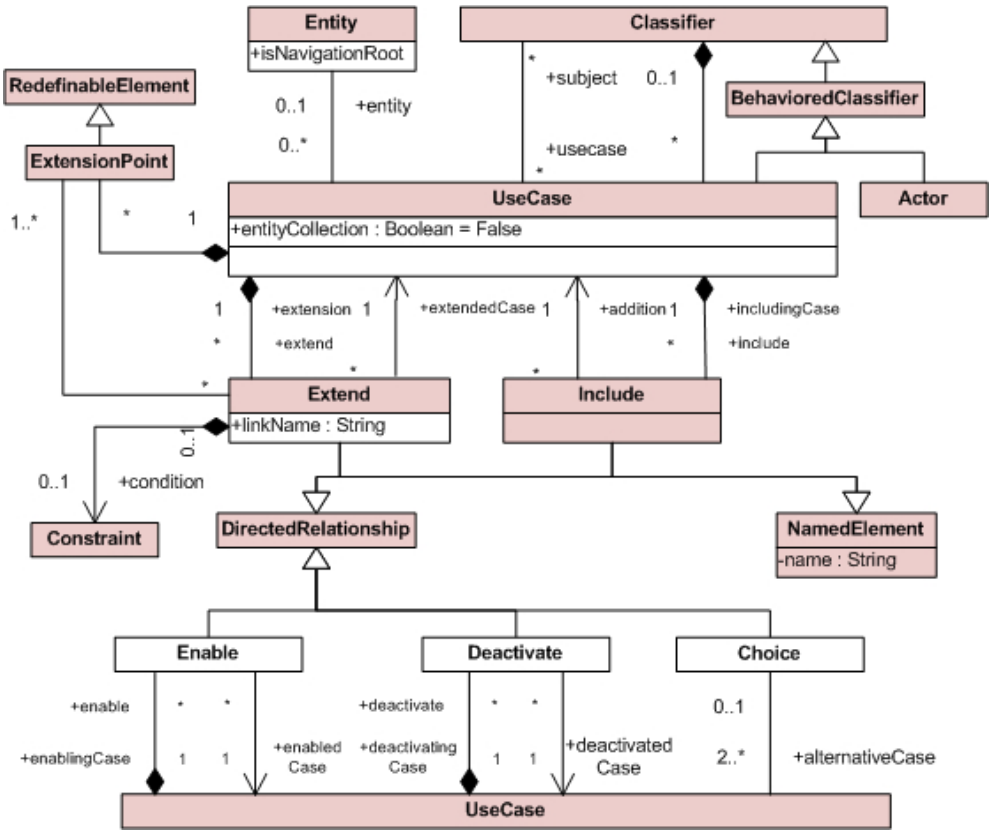

Figure 4.10: Metamodel for Use Case Models (Use case relations).

UseCase

It is the UML UseCase with added attributes that enable a smooth integration between a UCM and the respective DM. A UseCase may identify an entity class (BaseEntity or DerivedEntity) from the DM. If a BaseEntity is identified, then it is yet possible to restrict the admitted CRUD operations (Create, Retrieve, Update, Delete) available within the use case, by associating only the allowed CRUD operations to the UseCase. If entityCollection is set to true, the use case involves listing instances from the associated entity, and in this case, if the associated entity is a BaseEntity, it is possible to associate the use case to the Update operation of an associated entity (see section 4.5.2).

If the modeler wants to associate to the use case a user defined operation, then only one operation can be associated and the operation must be defined within the context of the associated BaseEntity. See below the possible kinds of use cases.

Extend

It's a merge increment to the standard UML extend relation between two use cases, to which one can associate a link name.

Operation

As already described for the DMM in the previous subsection.

4.5.2 Kinds of Use Cases

Although the metamodel doesn't explicitly differentiate them, one can distinguish two categories of use cases, which are not subclasses of UseCase in the metamodel, but rather state patterns that can be identified in use cases:

- 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. An independent use case satisfies:

```
not Extend.allInstances() −>exists(ext | ext.extension = self) and
not Include.allInstances() −>exists(inc | inc.addition = self) and
Relationship.allInstances() −>exists(rel | rel.relatedElement.include(self) and
rel.relatedElement −>exist(el | el.oclIsKindOf(Actor)))
```

- 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 optional or mandatory nature, respectively:

```
(Extend.allInstances() −>exists(ext | ext.extension = self) or
Include.allInstances() −>exists(inc | inc.addition = self)) and
not Relationship.allInstances() −>exists(rel | rel.relatedElement.include(self) and
rel.relatedElement −>exist(el | el.oclIsKindOf(Actor)))
```

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

- List Entity: view the list of instances of an entity (usually only some attributes, marked as identifying attributes, are shown)

```
(self.entityCollection = true and self.entity −>notEmpty())
```

- Create Entity: create a new instance of an entity

```
(self.entityCollection = false and self.entity −>notEmpty() and self.associatedOp −>forall(op
| op.oclIsKindOf(CreateOp)) )
```

- 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.

```
self.entity ->notEmpty() and
self.associatedOp ->size()=1 and
self.associatedOp ->forall(op | op.IsStatic())
```

The types of dependent use cases that can be defined in connection with the DM 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)

```
(self.entityCollection = false and
self.entity ->notEmpty() and
self.associatedOp ->forall(op | self.entity.ownedOperation.includes(op)) and
self.associatedOp ->forall(op | op.oclIsKindOf(RetrieveOp) or op.oclIsKindOf(UpdateOp)
or op.oclIsKindOf(DeleteOp)) )
```

- 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

```
(self.entityCollection = false and
self.entity ->notEmpty() and
self.associatedOp ->forall(op | self.entity.ownedOperation.includes(op)) and
self.associatedOp ->forall(op | not op.oclIsKindOf(CRUDop)) )
```

- 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

```
(self.entityCollection = true and
( Include.allInstances() ->exists(inc | inc.addition = self and Association.allInstances
->exists(a | a.memberEnd.includes(self.entity) and a.memberEnd.includes(inc.includingCase.entity)))
or
Extend.allInstances() ->exists(ext | ext.extension = self and Association.allInstances
->exists(a | a.memberEnd.includes(self.entity) and a.memberEnd.includes(ext.extendedCase.entity)))
))
```

- 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)

```
(self.entityCollection = false and
self.entity ->notEmpty() and
self.associatedOp ->forall(op | self.entity.ownedOperation.includes(op)) and
self.associatedOp ->forall(op | op.oclIsKindOf(CreateOp)) and
Extend.allInstances() ->exists(ext | ext.extension = self and Association.allInstances
->exists(a | a.memberEnd.includes(self.entity) and a.memberEnd.includes(ext.extendedCase.entity)))
)
```

- 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)

```
(self.entityCollection = false and
self.entity ->notEmpty() and
self.associatedOp ->forall(op | self.entity.ownedOperation.includes(op)) and
self.associatedOp ->forall(op | op.oclIsKindOf(RetrieveOp) or op.oclIsKindOf(UpdateOp)
or op.oclIsKindOf(DeleteOp)) and
Extend.allInstances() ->exists(ext | ext.extension = self and Association.allInstances
->exists(a | a.memberEnd.includes(self.entity) and a.memberEnd.includes(ext.extendedCase.entity)))
)
```

- 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)

```
(self.entityCollection = true and
( Include.allInstances() ->exists(inc | inc.addition = self and Association.allInstances
->exists(a | a.memberEnd.includes(self.entity) and a.memberEnd.includes(inc.includingCase.entity)))
or
Extend.allInstances() ->exists(ext | ext.extension = self and Association.allInstances
->exists(a | a.memberEnd.includes(self.entity) and a.memberEnd.includes(ext.extendedCase.entity)))
) and
self.associatedOp ->forall(op | self.entity.ownedOperation.includes(op)) and
self.associatedOp ->forall(op | op.oclIsKindOf(LinkOp)) )
```

- 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)

```
(self.entityCollection = true and
( Include.allInstances() ->exists(inc | inc.addition = self and Association.allInstances
->exists(a | a.memberEnd.includes(self.entity) and a.memberEnd.includes(inc.includingCase.entity)))
or
Extend.allInstances() ->exists(ext | ext.extension = self and Association.allInstances
->exists(a | a.memberEnd.includes(self.entity) and a.memberEnd.includes(ext.extendedCase.entity)))
) and
self.associatedOp ->forall(op | self.entity.ownedOperation.includes(op)) and
self.associatedOp ->forall(op | op.oclIsKindOf(LinkOp)) )
```

- 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)

```
(self.entityCollection = true and
( Include.allInstances() ->exists(inc | inc.addition = self and Association.allInstances
->exists(a | a.memberEnd.includes(self.entity) and a.memberEnd.includes(inc.includingCase.entity)))
or
Extend.allInstances() ->exists(ext | ext.extension = self and Association.allInstances
->exists(a | a.memberEnd.includes(self.entity) and a.memberEnd.includes(ext.extendedCase.entity)))
) and
self.associatedOp ->forall(op | self.entity.ownedOperation.includes(op)) and
self.associatedOp ->forall(op | op.oclIsKindOf(UnlinkOp)) )
```

Operations are user-defined in an Action Semantics-based action language, and supplement the basic CRUD instante operations (Create, Retrieve, Update and Delete) that are defined by default in every BaseEntity.

4.5.3 Use Case relations

The defined metamodel for use cases comprises the following use case relations:

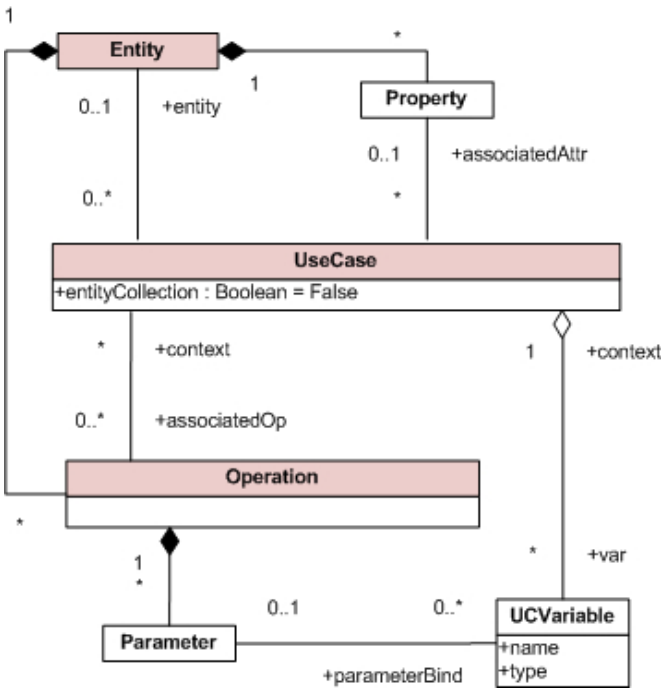

Figure 4.11: Metamodel for Use Case Models (use case associations to DM classes).

- Extension: As aforementioned it corresponds to the standard UML use case extension mechanism with one added meta-attribute.
- Inclusion: It corresponds to the standard UML use case inclusion mechanism. The including use case has full access to the included use case features, in particular the DM features associated to the use case.
- Inheritance: It is the standard UML use case inheritance mechanism. A use case that inherits from another use case, inherits all its features (included use cases, associated DM features, etc.).
- Enabling: The proposed UCM metamodel extends the UML with use case relations that recall HCI's task models [PMM97]: enable; deactivate; and choice. The enabling relation may be defined between two use cases included within another use case that sets a common context. Only when the enabling use case is performed, the enabled use case may be performed by the actor accessing them.
- Deactivation: The deactivation relation may be defined between two use cases included within another use case that sets a common context. The execution of the deactivating case disables the deactivated one.

- Choice: The choice relation enables the definition of alternative use cases. By performing one of the choice related use cases, the actor disables all the others.

Fig. 4.12 shows the types of relationships that can be defined among use cases constrained by DM's entities relations.

Task-model-like relations between use cases allow to further increase the expressive power of the proposed approach, and thus amplify the flexibility of the automatically generated UIs, giving the modeler more autonomy in relation to what can be modeled within a use case model. Figure 4.13 shows the syntax for use case model's enable and deactivate relations.

Figure 4.14 shows the possible options for choice relations' syntax.

From the defined metamodel, one can see that a use case may be associated to a base or derived entity, an entity property, an entity operation or a use case variable. This allows the modeler to define use cases that are at different abstraction levels. For example, one can define an enable relation between use cases associated to an entity and to allowable CRUD operations, but it is also possible to have an enable relation between use cases associated to entity features stating, for instance, that some operation can only be performed after seting the value of a given entity property or use case parameter.

In fact, a use case can be innerly defined by including use cases, whether it is associated to a domain base or derived Entity or not. The use case that is defined at the expenses of included cases associated to entity properties or use case variables is called an aggregator use case. Just like with include and extend relations, the concrete notation for task model like relations makes use of proper stereotyping of use case relations with "enable", "deactivate" or "choice". At its lowest abstraction level, this kind of use cases, together with the properly stereotyped use case relations, allow the modeler to define which set of attributes must be set first, and which depend on other attributes, or are deactivated by setting other attributes. At this modeling level, it is possible to associate an included use case to a class' attribute, user-defined operation, or CRUD operation, or it can be a use case variable, which is a kind of use case that may be associated to an operation parameter accessible from the aggregator use case. It is allowed to have an aggregator UC, associated or not to any class, and have several included UCs associated to different entity classes. The entity, operation(s), and link type (when needed) associated to each use case are, as explained before, specified by using tagged-values.

Figure 4.15 shows an example of using task-model-like relations for detailing a use case, by relating included use cases associated to entity attributes or operations.

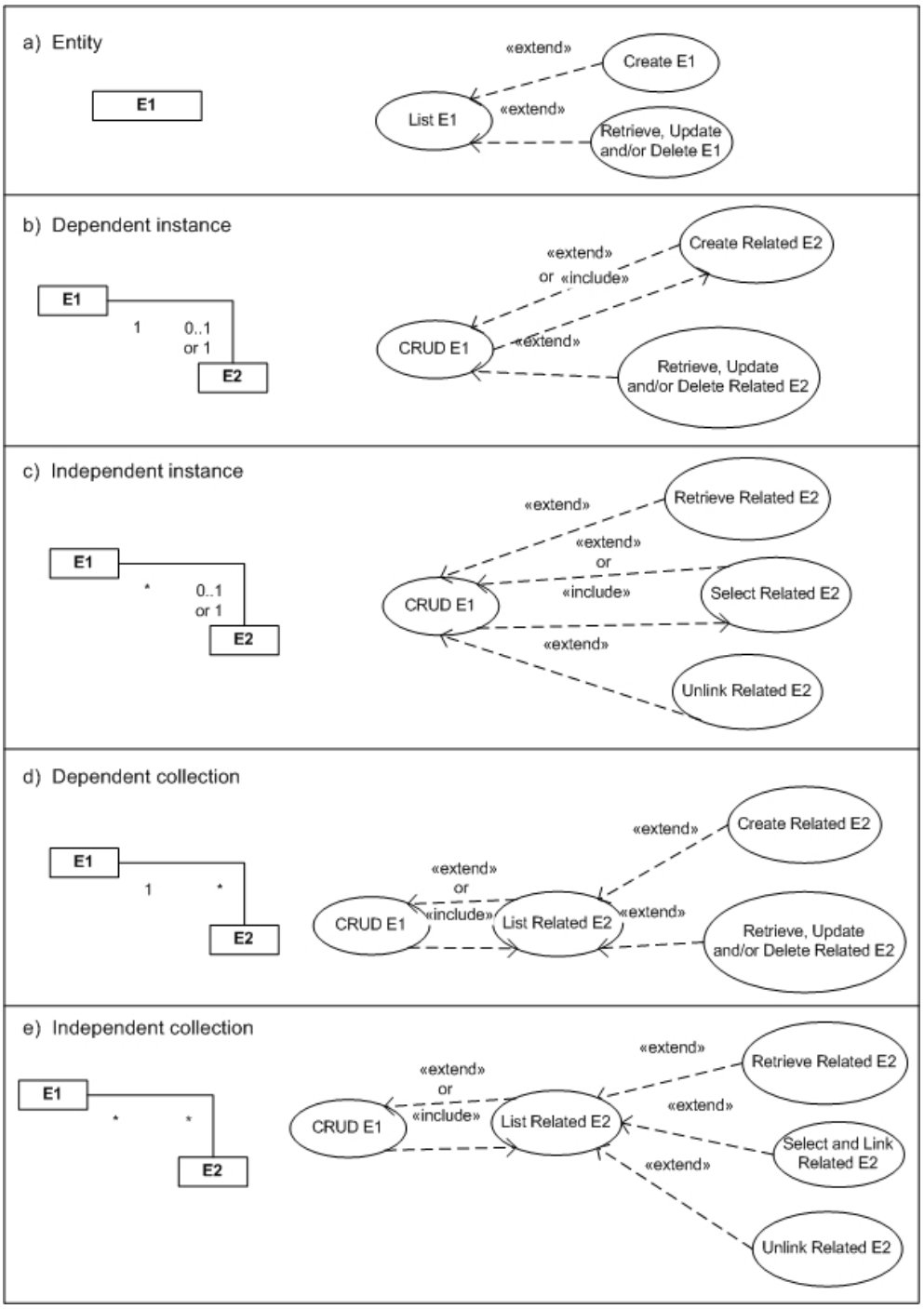

Figure 4.12: Possible types of relationships among use cases for different domain model fragments (note: aggregations and compositions impose, on the UCM, similar constraints as simple associations).

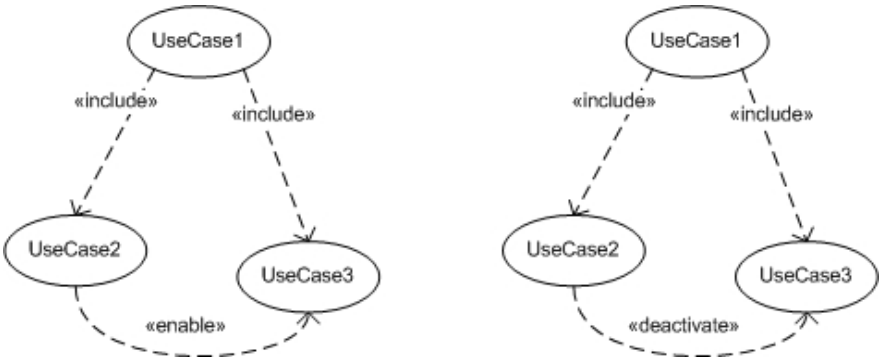

Figure 4.13: Enable and deactivation relations between two use cases.

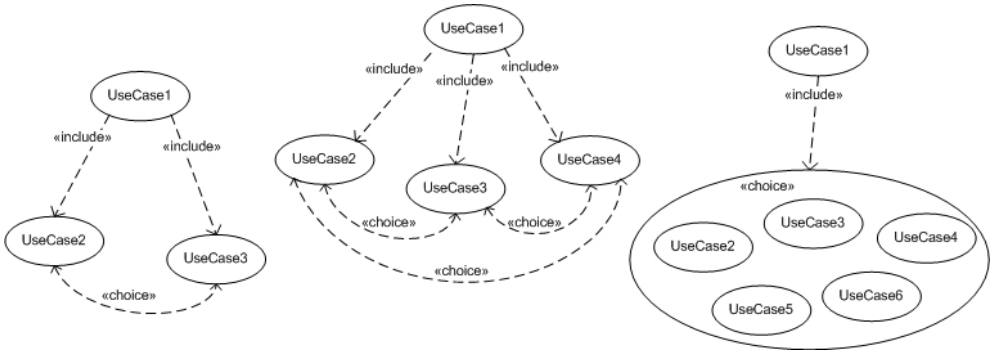

Figure 4.14: Choice relation between two or more use cases.

4.5.4 Example

Fig. 4.16 illustrates an example of a use case model that conforms to the presented use case metamodel.

The example shows us two actors with different system usage profiles, and so different available functionality on the system. The Librarian directly accesses four use cases: "List Books", "List Loans", "Add Loan" and "Register New Loan". The first two are List Entity use cases and, from each of them, it is possible to add or edit the selected item through CRUD entity use cases (e.g.: "Add Loan", "Edit Book").

Use cases "Add a new Book" and "Edit Book" include a list of related bookcopy entities ("List BookCopy"), from where it is possible to add and edit bookcopies related to the selected book.

Add and Edit Loan use cases include "Select BookCopy" and "Select Borrower" use cases, which enable the user to select a bookcopy and a borrower when creating or updating a loan.

Use case "Register New Loan" acts as an entry point for the user to choose between "Register Loan to existing Borrower" and "Register Loan to new Borrower". These use cases extend the first one. "Register Loan to existing Bor-

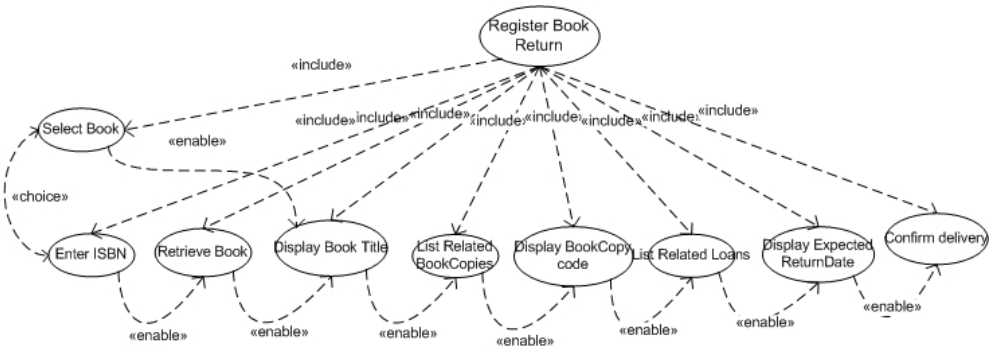

Figure 4.15: Use case innerly defined through included use cases associated to lower-level domain model elements (attributes and operations), which are related among each other through task-model-like relations.

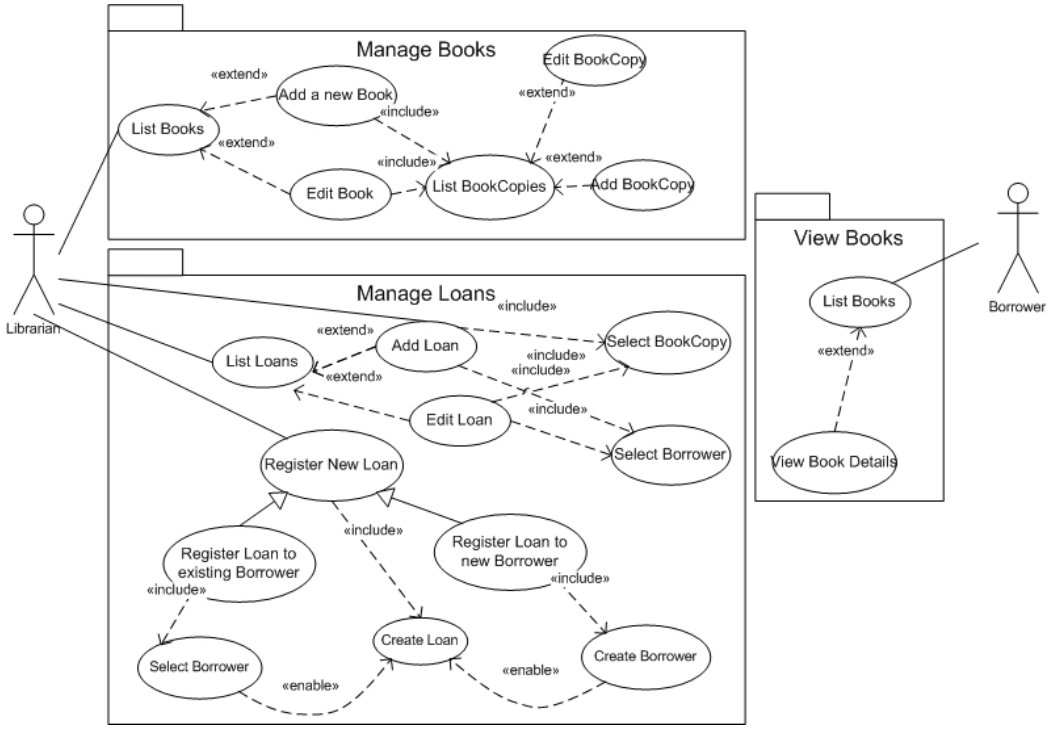

Figure 4.16: Partial use case model (UCM) for the Library Management System.

rower" includes a use case to select a borrower from a list of existing borrowers, and use case "Register Loan to new Borrower" includes use case "Create Borrower". Both use cases ("Select Borrower" and "Create Borrower") enable use case "Create Loan", which is also present by inheritance of included use cases.

In order to keep the integrity between the domain model and the use case model, a set of tagged values, corresponding to the meta-attributes of UCMM,

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

Table 4.3: Entities and operations associated (via tagged values) with the use cases in Fig. 4.16.

shall be defined. Table 4.3 shows the tagged values for the presented LibrarySystem example.

4.6 Metamodel for User Interface Models

The metamodel package for User Interface models (UIMM) is depicted in Figures 4.17 and 4.18. The UIMM is based on MOF and imports some of the elements defined in the DMM.

The top level element from which every element in the UIMM inherits from is AbstractInteractionObject, or AIO. There are two types of AIO: ComplexAIO and SimpleAIO. The first models elements that contain other elements, and the latter models elementary objects used within complexAIO elements. In this section, the main UIMM elements are introduced.

4.6.1 Main metamodel elements

InteractionSpace

An InteractionSpace represents an abstract object that, at PIM level, is a UI container where interaction occurs. It serves the purpose of interfacing the interaction between the user and the system. An InteractionSpace is composed of InteractionBlocks and may be directly instantiable or it may be specialized as:

- ActorMainSpace: May be used to model an interaction space with a menu bar. A menu bar is composed of menus, each of which may comprise several

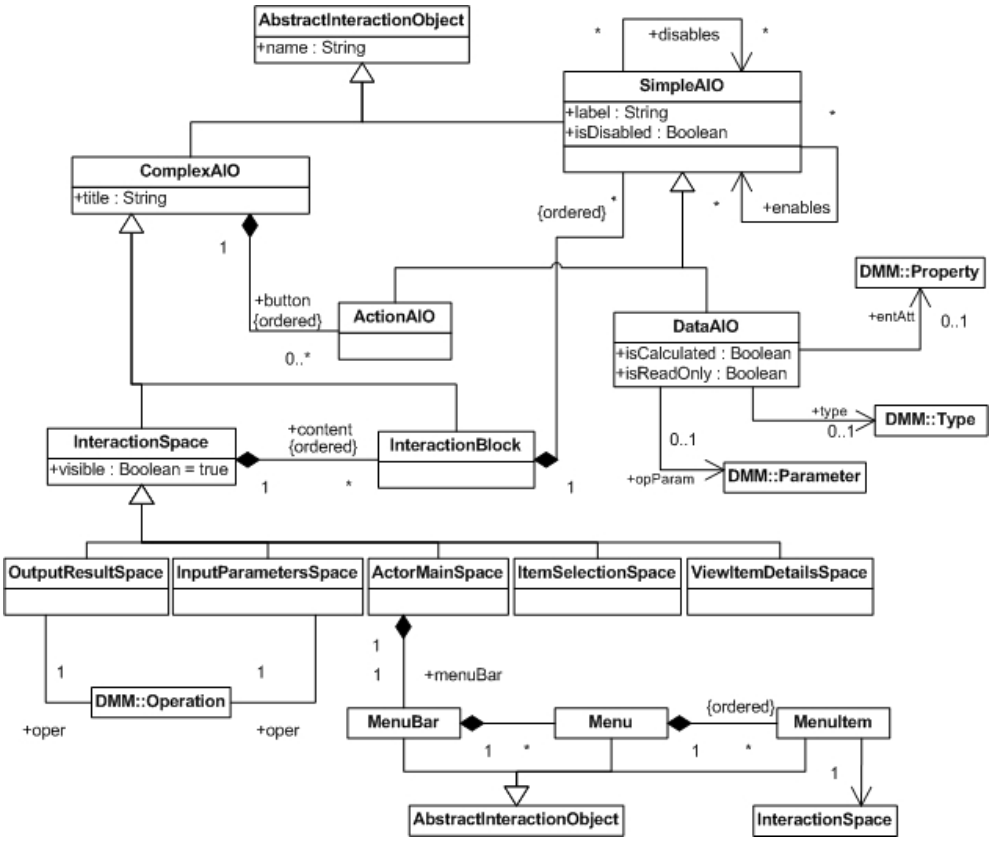

Figure 4.17: Metamodel for User Interface Model.

menu items, from where navigation to other interaction spaces is possible. It aims to be the entry point in the application, for a given actor.

- InputParametersSpace: It is an interaction space that serves the purpose of collecting the values of an operation's parameters, and than trigger the operation passing the parameters to it.

- OutputResultSpace: It is an interaction space that aims to notify the user about the success or failure of an operation execution. If notifying the success of the operation, and if it returns a result, then the result may also be shown to the user.

- ItemSelectionSpace: Interaction space that may include a ViewList or ViewRelatedList block, and is used to return a reference to the item in the selected line.

- ViewItemDetailsSpace: Interaction space that must include a ViewEntity block, and forces all its DataAIO elements to be read-only.

InteractionBlock

An InteractionBlock is a container of SimpleAIOs. It may be directly instantiable or it may be specialized as:

- ViewEntity: Block, associated to an entity in the domain model, that may contain DataAIOs, intended for inputting (navigation through a ToCreateInstance navivation ActionAIO), displaying (navigation through a ToViewInstance navivation ActionAIO, or marked as readOnly) or editting (navigation through a ToUpdateInstance navigation ActionAIO) attribute values.

- ViewList: Block, associated to an entity in the domain model, that may contain DataAIOs, intended as columns in an output only list of attribute values.

- ViewRelatedEntity: This interaction block has a structure similar to a ViewEntity block, but it must be in an interaction space containing also a ViewEntity. The ViewEntity must be associated to an entity that has a to-one relation to the entity associated to the ViewRelatedEntity.

- ViewRelatedList: This interaction block has a structure similar to a ViewList block, but it must be in an interaction space containing also a ViewEntity. The ViewEntity must be associated to an entity that has a to-many relation to the entity associated to the ViewRelatedEntity. If isCollapsable is true, the ViewRelatedList must have a collapse/expand feature.

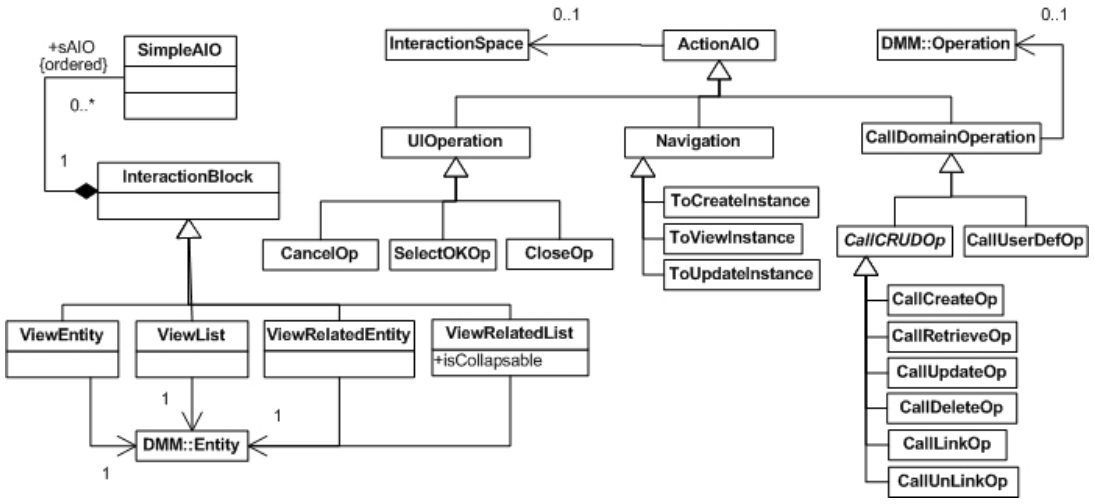

Figure 4.18: Metamodel for User Interface Model - InteractionBlock relations and subtree, at the left, and ActionAIO subtree, at the right.

DataAIO

A SimpleAIO may be a DataAIO or an ActionAIO. DataAIOs are typically associated to entity attribute properties in the DM, provided they are included within an interaction block specialization associated to a domain entity. Alternatively, a DataAIO may be associated to an operation parameter, for inputting a value, or it may be just associated to a domain model Type. The latter case may be used when willing to show the result of an operation execution, or when inputting a value to a variable that may be bound to an operation parameter afterwards. If associated to a domain attribute, a DataAIO may be intended for input a value, display a value or edit a value (display and allow modification). When associated to an attribute, the type must be equal to the type of the attribute. In particular, when the type is an enumeration, the concrete notation of the UIM will be different, just as the code that may be generated from it.

ActionAIO

ActionAIOs may be *CallDomainOperation*s, like the call of a CRUD or user defined operation, *Navigation* operations, that allow navigation between interaction spaces, or *UIOperation*s, that allow action over UI elements (eg.: closing a window). A CallDomainOperation ActionAIO may be:

- A CallUserDefOp, that is associated to a user defined operation in the Domain Model;
- A specialization of the abstract meta-class CallCRUDOp:
  - CallCreateOp, that must be associated to DMM::CreateOp.
  - CallRetrieveOp, that must be associated to DMM::RetrieveOp.
  - CallUpdateOp, that must be associated to DMM::UpdateOp.
  - CallDeleteOp, that must be associated to DMM::DeleteOp.
  - CallLinkOp, that must be associated to DMM::UpdateOp.
  - CallUnLinkOp, that must be associated to DMM::UpdateOp.

By using OCL, Table 4.4 shows the invariants defined over the UIMM.

4.6.2 User Interface Model concrete representation

The above described metamodel for defining models of the user interface specifies the abstract syntax of the UI modeling language. If the UIM were to be generated only as an intermediate artifact towards the final concrete UI, this would be enough. But, if one wants the modeler to be able to build from scratch a UIM,

Context Class	Invariant
ViewEntity, ViewList, ViewRelatedEntity, ViewRelatedList	((not self.sAIO.entAtt −>isEmpty()) implies self.entity = self.sAIO.entAtt.entity) AND ((not self.sAIO.operation −>isEmpty()) implies self.entity = self.sAIO.operation.entity)
DataAIO	(not self.entAtt −>isEmpty()) implies self.type = self.entAtt.type

Table 4.4: UI Metamodel invariants.

and to modify an automatically generated UIM, it is needed a concrete language syntax for defining a UIM.

This way, as the UML support for user interfaces is not effective [Nun01, Pin00], the adopted syntax is Larry Constantine's Canonical Abstract Prototypes (CAP). Refer to section 2.8.1 and [Con03].

InteractionSpace and InteractionBlock

Figure 4.19 shows the mapping between InteractionSpace and InteractionBlock abstract elements and its corresponding concrete canonical notation.

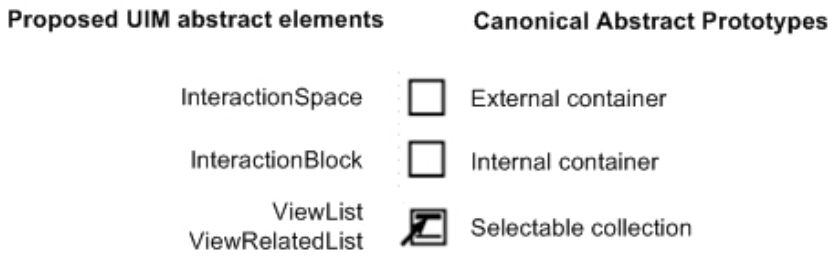

Figure 4.19: Canonical abstract notation for interaction spaces and interaction blocks.

From the metamodel, one can see that an interaction space is composed of interaction blocks. All interaction spaces have the same canonical concrete representation, while for interaction blocks this is not true. In fact, there are interaction bloks where contained DataAIOs are interpreted as fields, and others where DataAIOs are interpreted as data columns. For the first ones, ViewEntity and ViewRelatedEntity, the concrete notation is as for general InteractionBlocks. In the latter case, ViewList and ViewRelatedList, the concrete notation is as shown in Fig. 4.19.

An ActorMainSpace is an interaction space that may comprise a MenuBar. Fig. 4.20 shows the concrete notation for MenuBar, Menu and MenuItem. Other

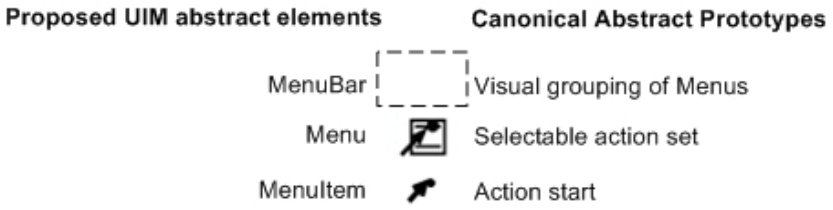

Figure 4.20: Canonical abstract notation for Menus.

interaction spaces, although having the same concrete notation, have strong constraints over the elements they may comprise. Besides ActorMainSpace, this is the case of ItemSelectionSpace, ViewItemDetailsSpace, and InputParametersSpace and OutputResultSpace, which are addressed below (see subsection "Other specializations of InteractionSpace").

DataAIO

The concrete canonical notation mapped from DataAIOs distinguishes the ones that are only for data input, from the ones intended for editing and from output-only data AIOs (see Fig. 4.21).

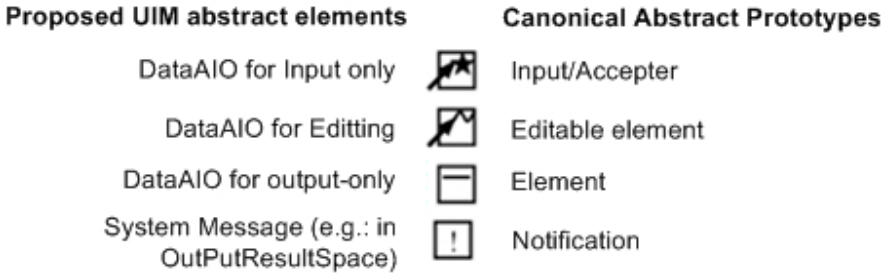

Figure 4.21: Canonical abstract notation for DataAIOs.

There is also a concrete notation for automatic system notification messages, that are used by the system for notifying about the success or not of an operation execution.

ActionAIO

Fig. 4.22 shows the concrete canonical elements for different types of ActionAIOs.

Other specializations of InteractionSpace

Besides the aforementioned interaction spaces, there are also the following ones, which may be considered as "pre-built" UI model constructs:

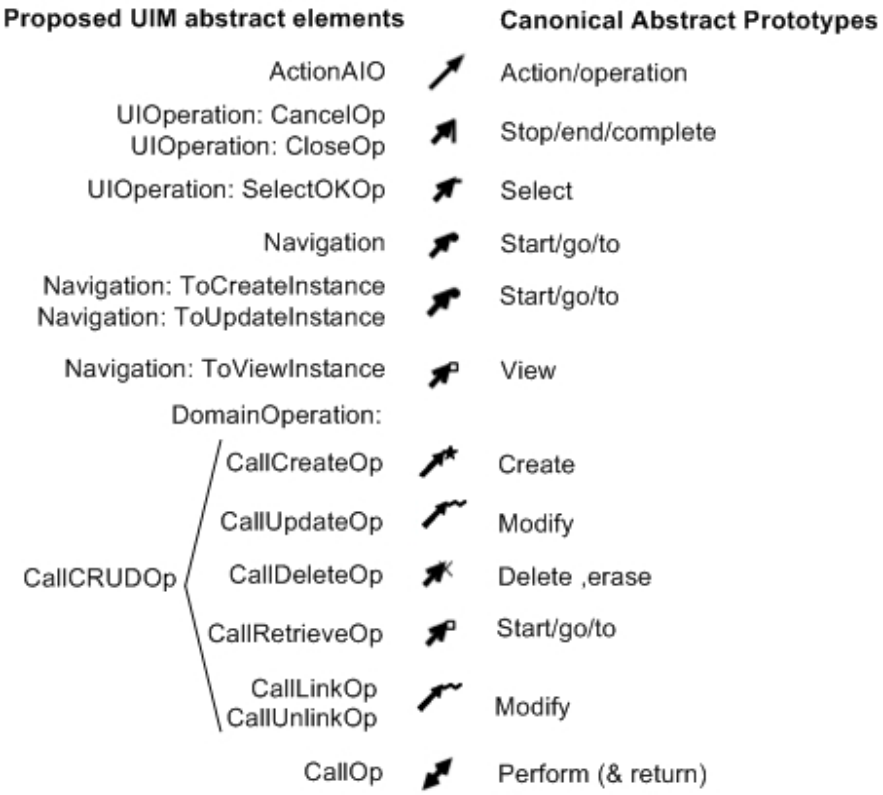

Figure 4.22: Canonical abstract notation for ActionAIOs

- ItemSelectionSpace: It is a "pre-built" UI model element that returns a reference to the selected item to the block to which it returns (see Fig. 4.23).

- ViewItemDetailsSpace: It is a "pre-built" UI model element that receives an instance reference and displays the values of its attributes (see Fig. 4.23).

- InputParametersSpace and OutputResultSpace: Are "pre-built" UI model elements associated to a domain model operation (see Fig. 4.24). An InputParametersSpace is intended to accept the parameters values for passing to a DM operation, invoke it and wait for its completion, and then display a notification of success or failure, about the operation execution, along with the operation's result, in case one exists, through an OutputResultSpace.

Further notes about the abstract UIM to CAP mapping

When building a concrete domain model or use case model, the modeler will need to assign values to the meta-attributes, that is the attributes of the elements in the metamodel. In the DM and the UCM, this is done through tagged-values.

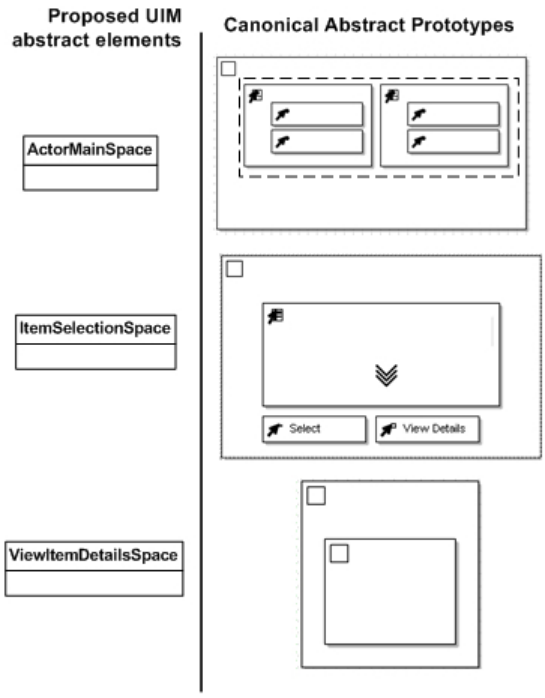

Figure 4.23: Canonical abstract notation for "pre-built" interaction spaces.

Also when building a concrete UIM, the modeler will need to assign values to the UI meta-attributes. For this to be possible, Canonical Abstract Prototypes' tools would need to be extended with tagged-values features.

4.6.3 Example

Figures 4.25 and 4.26 illustrate an example of an abstract and concrete user interface model, respectively, conforming to the presented user interface metamodel.

Fig. 4.26 represents part of the UIM elements of fig. 4.25. Indeed, they represent a possible partial UIM for the LibrarySystem example that has accompanied us.

Each actor has its own unique main space, that corresponds to the actor's entry point when interacting with the application. The Librarian actor main space, in the figures, is composed of a MenuBar with two Menu elements. Each menu element has an ordered collection of menu items, from where the actor may navigate to other interaction spaces.

From the example abstract UIM (Fig. 4.25), one can see that the librarian may navigate to the "Add Loan" interaction space by following one of two possible navigation paths:

- Directly from his/her main space, by selecting menu item "Add Loan" (also represented in the concrete UIM in fig. 4.26); or,

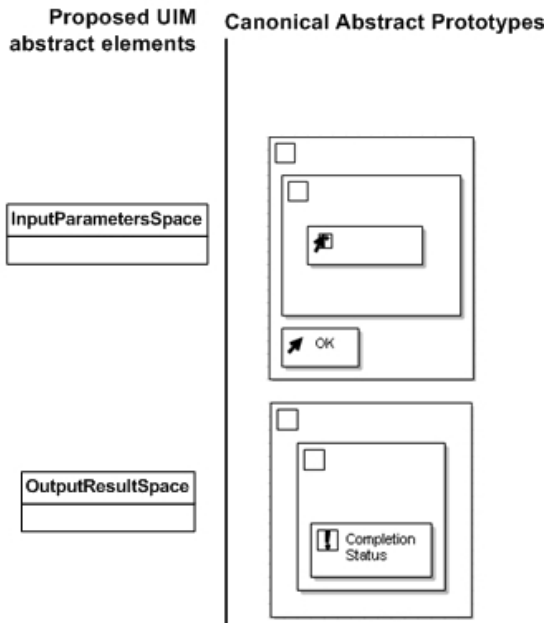

Figure 4.24: Canonical abstract notation for "pre-built" interaction spaces.

- By selecting "List Loans" in the menu, which leads him to a ListLoans interaction space (represented in the abstract UIM). The ListLoans interaction space comprises a ViewList interaction block and two navigation action AIOs (AddLoan: ToCreateInstance and EditLoan:ToUpdateInstance). By selecting the AddLoan actionAIO he will navigate to the "Add Loan" interaction space.

The "Add Loan" interaction space comprehends three interaction blocks: a ViewEntity block, that allows the input of Loan's attributes dueDate and effectiveReturnDate (although this last one should be left unfilled), and two ViewRelatedEntity blocks, for displaying the identifying attributes of the related BookCopy and Borrower instances, each one comprising a navigation actionAIO for navigating to ItemSelectionSpace instances that enable creation or modification of the related instances.

4.7 Conclusions

An interactive system development process has been defined, based on a system specification according to three points of view: structural view (domain model); functional view (use case model); and, user interface view (user interface model). The proposed process generates a UIM from the DM and the UCM.

The metamodels for the DM, UCM and UIM have been formalized, and a concrete notation for UIMs has been defined.

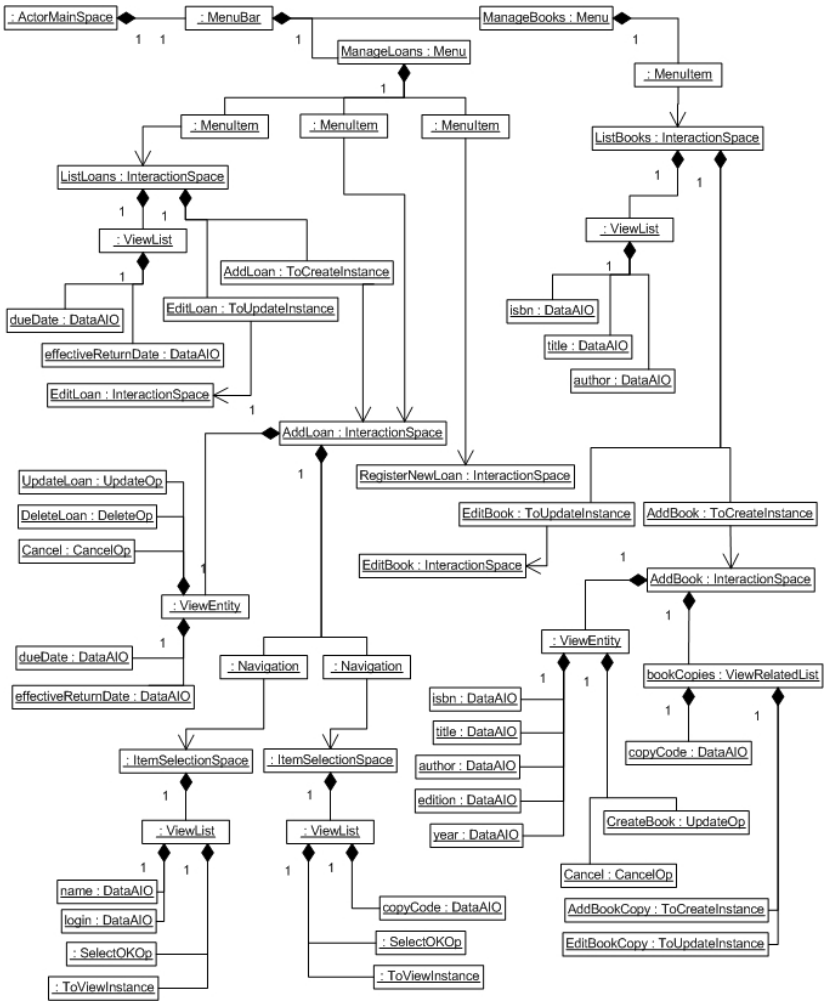

Figure 4.25: Partial UIM abstract syntax elements for the Library Management System.

In the next chapter, a set of model transformation rules will be defined, enabling the UIM generation from the DM alone or from an integrated pair of models comprising a UCM and a DM.

The defined metamodels allow the modeler to use advanced features and to make use of a combination of elements only restricted by the metamodels constraints (including the ones that are inherited from the UML metamodel). Not all of the possible element patterns will have a defined transformation rule in the next chapter. This means that not all patterns from the DM and UCM will generate elements in the UIM, because no transformation rule is defined for them. Hence, if a modeler wants to take advantage of the M2M transformation processes he will need to be more restrictive than what the metamodels impose.

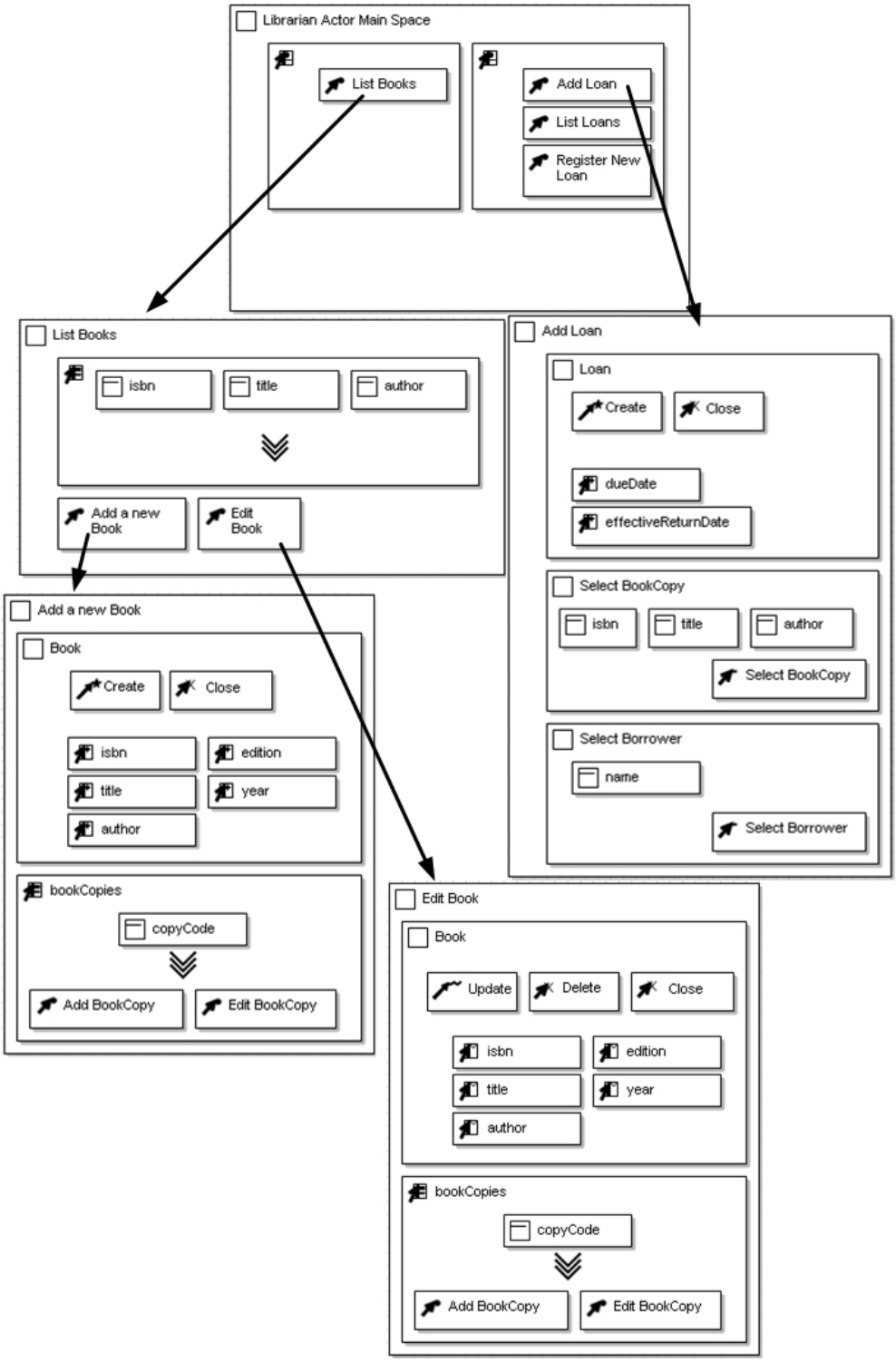

Figure 4.26: Partial UIM concrete abstract prototypes for the Library Management System.

Chapter 5

Model-Transformation Rules

This chapter presents the mapping rules between metamodel elements that allow the model-to-model transformations that enable the generation of a use case model from a domain model and a user interface model from domain and use case models. The rules are presented together with a running example.

5.1 Introduction

The metamodels for specifying a data-intensive forms-based software application, presented in the previous chapter, allow the development of a rigorous interactive system model. In fact, a modeler may use those metamodels to specify an interactive system even if he isn't thinking of following the recommended development process or of using automatic generation processes (refer to section 4.2). Nevertheless, if the modeler is thinking of following the recommended development process, and thus make use of automatic model transformation and code-generation processes, then he will need to keep the developed models fully integrated, and will need to be more restrictive in what concerns to some of the constraints imposed to the domain and use case models.

In this chapter, the transformation rules defined between the metamodels, that allow a model-to-model automatic transformation between metamodel instances, will be addressed.

As explained in section 4.1, the automatic interactive application generation, from domain and use case models, is based on a two-step approach. In the first step, a UI model is generated by applying a set of model transformation rules to the domain model alone, DM2UIM model-transformation process, or to the integrated domain and use case models, DM+UCM2UIM model-transformation process (recall Fig. 4.1).

After the model-transformation process (see DM2UIM or DM+UCM2UIM in Fig. 5.1), which generates a default UI model for the interactive application being modeled, a second step is undertook by putting into action a target platform dependent model-to-code (M2C) generation process. To this M2C process the DM

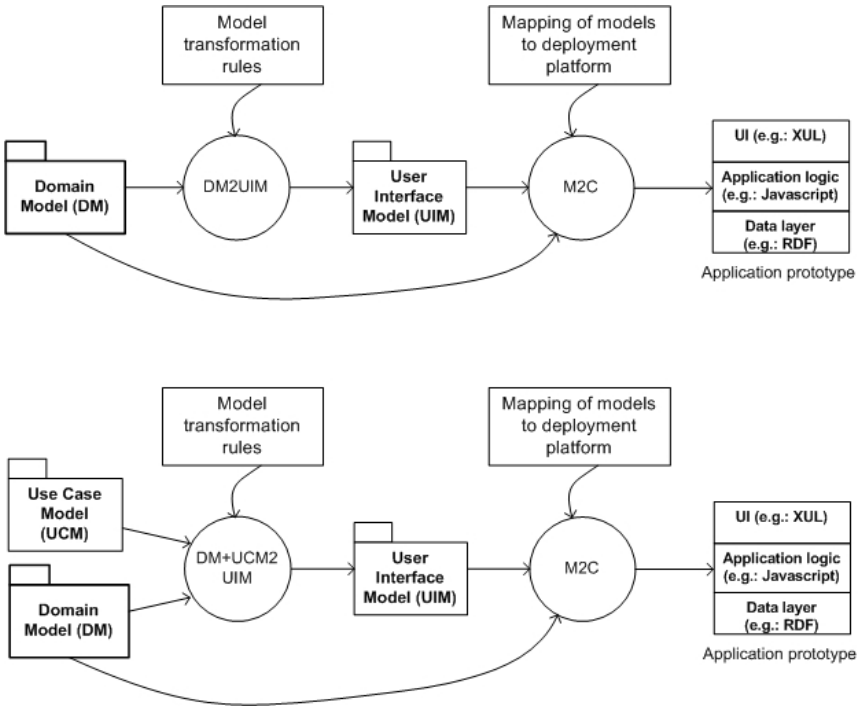

Figure 5.1: General approach to UI generation.

and the UIM are inputted, in order to generate the final code for the application or prototype.

In the scope of this PhD dissertation, a proof-of-concept tool has been developed (refer to chapter 6), in which the M2C process generates UI descriptions in XUL [Moz], business logic actions in Javascript, and data persisting structures in RDF.

The next sections address the transformation rules that configure processes DM2UIM and DM+UCM2UIM. It is to notice that the defined automatic transformation rules are more restrictive than what is possible to model by using the presented metamodels.

This chapter presents the transformation rules defined between:

- The Domain Metamodel and the UI Metamodel (DM2UIM), that enable a model-to-model transformation between a domain model and a user interface model;
- The Domain Metamodel plus the Use Case Metamodel and the UI Metamodel (DM+UCM2UIM), that allow a model-to-model transformation between a core system model, comprising a DM and a UCM, to a user interface model;
- The Domain Metamodel and the Use Case Metamodel (DM2UCM), that

allow a model-to-model transformation between a domain model and a use case model, in one of the early iterations of the process.

The model transformation strategy (refer to section 2.4.3) is based on the incremental application of transformation rules that generate or modify elements in the target model (the UIM). Each rule is defined with a left-hand-side (LHS), that defines a pattern in the source model and within the previously generated target model elements, and a right-hand-side (RHS) that specifies the elements of the target model that shall be generated or modified, together with their relation to elements in the source model. In the next subsections, the elements of the LHS are shaded, and the ones of the RHS are not. Moreover, at least one element of the LHS has a bold border, meaning that it is the pattern root, that is where the pattern matching engine will start its search, before trying to match the rest of the LHS elements.

The transformation rule execution mechanism searches the source model, and the already generated target model elements, for the LHS pattern. When a pattern matching occurs, the elements of the RHS are generated or modified according to the specified relationships to elements in the source model.

Together with a more formal presentation, a running example will be used to help unveiling the mapping rules defined. The transformation rules' presentation structure is as follows:

- Pattern and mappings: A pattern (LHS) of the source metamodel elements and of the previously generated target model elements, is presented, through a UML-like graphical notation and OCL constraints, together with mappings to the target metamodel elements being created or modified (RHS). The notation enables the specification the same classe playing different roles in the defined rule.

- Abstract syntax: It will be shown an object model illustrating the metamodels' instances for a part of the running example. Instances of DMM, UCMM or UIMM will be shown when relevant, illustrating the transformation relation being presented.

- Concrete syntax: The concrete syntax of the example will be shown for the DM, UCM and UIM (using CAP as presented in sections 2.8.1 and 4.6.2).

- Textual explanation: A textual explanation of each rule will be made, supported by the aforementioned running example.

Before closing the chapter, some conclusions are drawn from the results obtained.

5.2 Domain Model to User Interface Model Transformation Rules

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

Table 5.1 summarizes the main transformation rules for generating a UIM/UIP from a domain model. These were partially previously addressed in [dCF08], and shortly describe what one would like to have generated in a UI.

As the defended approach comprises the generation of an intermediate PIM level user interface model, the next sections address the transformation rules between the domain metamodel (DMM) and the UI metamodel (UIMM) following the presentation structure introduced before.

When the UIM is generated solely from the domain model, a special class with metamodel attribute "isNavigationRoot" set to true has to be created and linked to the domain classes that should correspond to the application entry points. A more flexible approach, starting from DM and UCM is explained in section 5.3.

As in the previous chapter, the LibrarySystem example will help better illustrate the transformation rules from a domain model (DM) to a user interface model (UIM). Recall Fig. 4.9, which depicts our Library System domain model.

In order to be able to identify the application user interface entry point, the DM is rooted in a class with the *isNavigationRoot* meta-attribute set to *True* (System in our example). 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.

The rules for the M2M process DM2UIM, which transforms elements in the DM into elements in the UIM, are the following, and its application follows the presented order:

1. DM2UIM01: Transform single base entities and primitive type attributes
2. DM2UIM02: Transform enumerated type properties
3. DM2UIM03: Transform inherited properties
4. DM2UIM04: Transform derived entities and derived attributes
5. DM2UIM05: Transform associations, aggregations and compositions
6. DM2UIM06: Transform user defined operations
7. DM2UIM07: Transform the navigation root

DM 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.
Derived attribute	Calculated output-only 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).
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.
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.
Class invariant	Validation rule that is called when creating or updating instances of the class.
Event-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 5.1: DM to UIM/UIP transformation rules.

Note that elements of the target model, generated in previously applied rules, may be part of the LHS in ulterior rules applications.

5.2.1 DM2UIM01: Transform single base entities and primitive type attributes

For each non-abstract base entity class with no subclasses, it is created an InteractionSpace with a ViewEntity block having DataAIOs, each one corresponding

to an entity property with a primitive type, and ActionAIOs for the CRUD operations on the entity. This rule is defined in figure 5.2.

For instance, for the class Book (see Figs. 4.9 and 5.2), it is created an interaction space with a ViewEntity, having a DataAIO for each entity attribute. This corresponds, in the UI prototype, to a form with a label and an input field for each entity attribute (attribute access modes are not being taken into account).

A base entity is assumed to have CRUD operations, as mentioned before (see chapter 4). This way, although the modeler doesn't specify the CRUD operations for a given base entity, they are assumed in the abstract syntax, and so are transformed as illustrated in Fig. 5.2.

The association from *BaseEntity* to *CreateOp*, *RetrieveOp*, *UpdateOp* and *DeleteOp* is made through the composite relation from *Entity* to *Operation* (recall the DMM in Figs. 4.7 and 4.8), meaning that the four $CRUDop$ subclasses, represented in the figure, belong to the same collection property (role), namely *ownedOperation*. The same is true for the composite association between $ViewEntity$ and the collection of $DataAIO$s, and *CreateOp*, *UpdateOp* and *DeleteOp*, in the RHS of the rule. They are all members of the $InteractionBlock$'s collection property in the composition from $InteractionBlock$ to $SimpleAIO$ (recall the UIMM in Figs. 4.17 and 4.18).

The multiplicities in the rules definition are the multiplicities of the pattern and of the generated elements, and so may be more restrictive than the ones defined for the metamodel.

The represented attributes, in each class, include the relevant inherited attributes. For instance, $DataAIO$ has attributes $isCalculated$ and $isReadOnly$, and inherits *name* and *label*. It also inherits $isDisabled$, which is not represented in the figure because in DM2UIM model transformation process all generated $SimpleAIO$s are enabled ($isDisabled = False$).

5.2.2 DM2UIM02: Transform enumerated type properties

Enumerated types are defined in the model as classes with an "enumeration" stereotype, and attributes of an enumerated type are class properties that are the association end of an association to an "enumeration" class. Fig. 5.3 illustrates the transformation rule for enumerated type entity attributes: a DataAIO is added to the view entity corresponding to the base domain entity in the LHS, and links to the domain property that generates it. To try to keep the figure simple, some attributes are not represented, but they have the same treatment as the preceding rule.

In Fig. 5.4, the concrete UI elements generated by the M2C process, and that have origin in a class relation to an "enumeration" class, can be seen in the BookCopy's form window. After the M2M transformation from the relation between

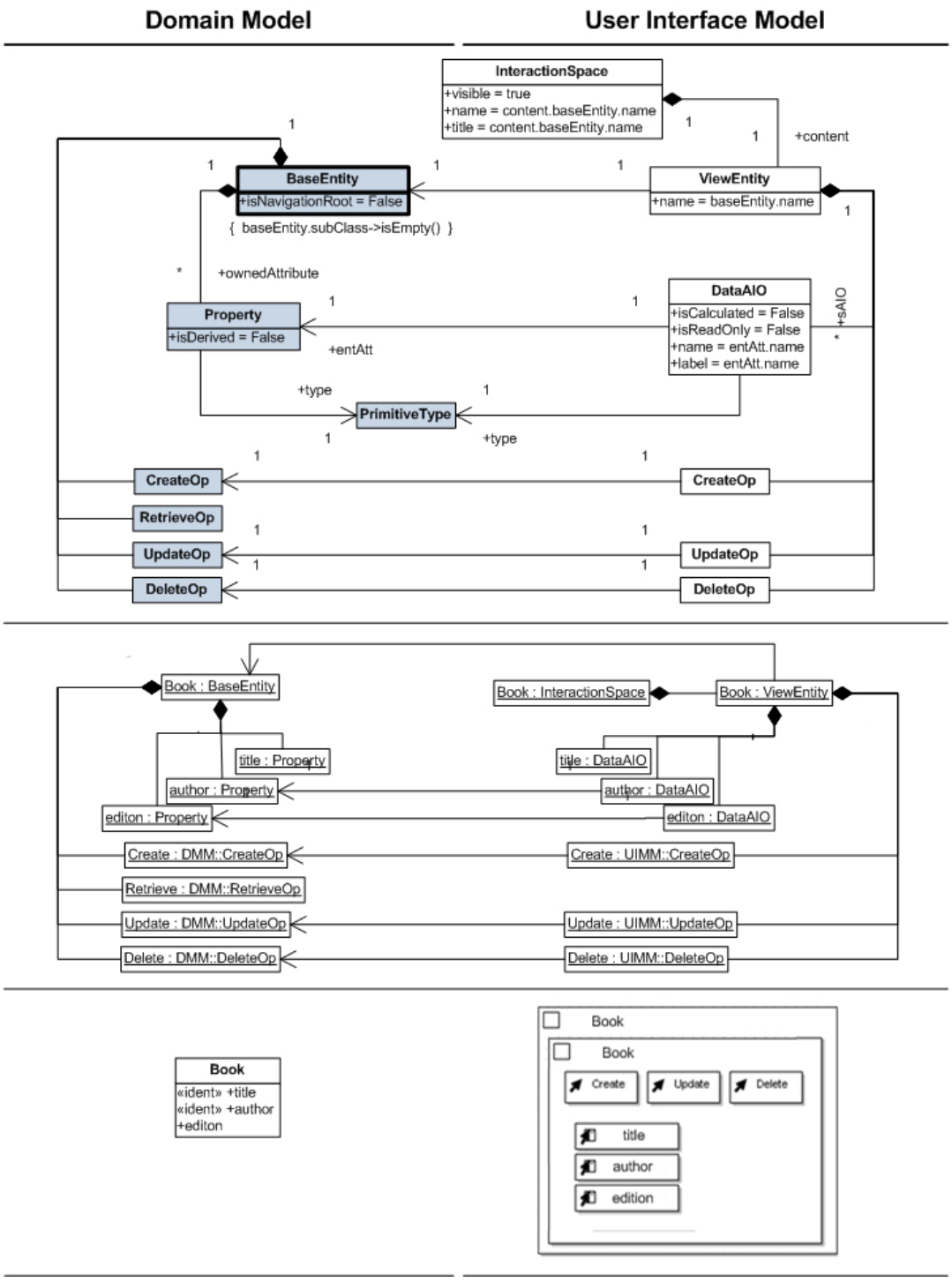

Figure 5.2: DM2UIM01: Mapping rule for transforming a single base entity, assuming CRUD operations, into an UI interaction space with convenient simple AIOs.

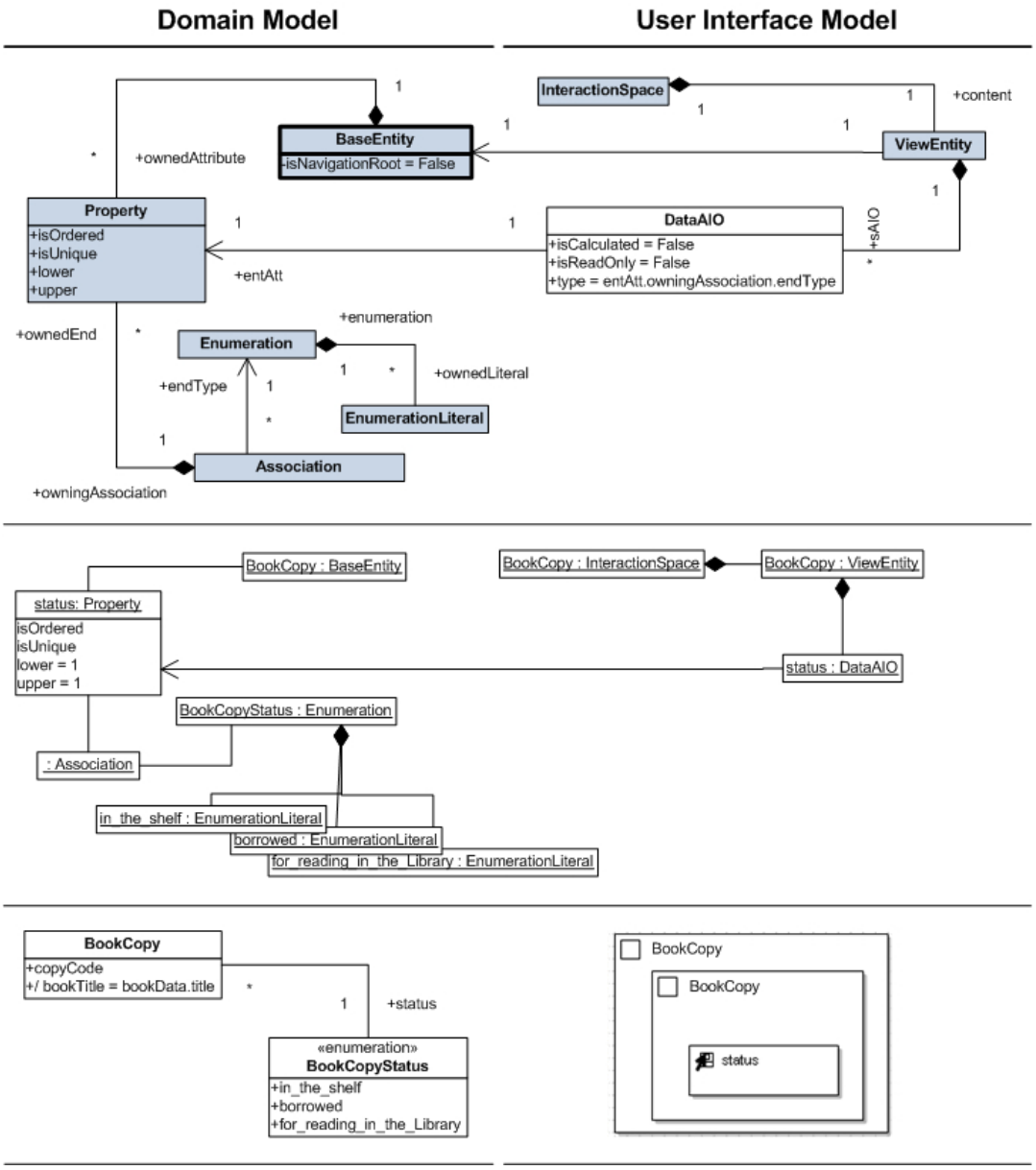

Figure 5.3: DM2UIM02: Transforming an enumerated type attribute to UI model elements.

class BookCopy and the enumerated type BookCopyStatus, explained above, the M2C process generates a list of radio buttons with the enumeration fields, in the BookCopy form, from the DataAIO that links to the domain enumerated type. The role name is used as an attribute, and one of the enumerated fields may be selected by the user through a radio button.

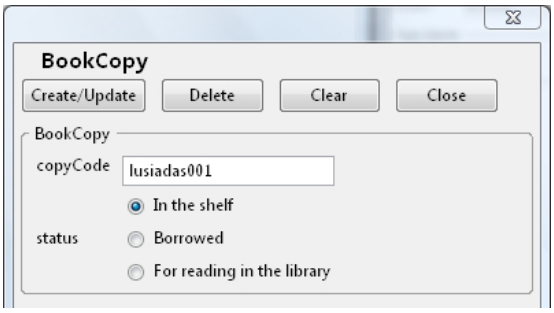

Figure 5.4: Partial BookCopy Form generated from the BookCopy interaction space, showing the elements generated from the relation to the BookCopyStatus enumerated type.

5.2.3 DM2UIM03: Transform inherited properties

In the proposed approach, only single inheritance is currently supported for automatic generation, and a transformation rule is defined only for the leaf classes of the inheritance hierarchy. Each leaf class inherits all the attributes and constraints from its ancestor classes, and so the DataAIOs corresponding to the inherited attributes are added to the previously generated InteractionSpace and ViewEntity block. Fig. 5.5 illustrates this transformation rule.

For searching for the LHS elements, first all base classes with no subclasses are searched. Then, for each one of those base classes, all the respective superclasses are collected, and all their properties are mapped to DataAIOs in the ViewEntity of the initial class.

The rule, represented in Fig. 5.5, depicts two *BaseEntity* class roles. One that is played by classes with no subclasses (on the right), and the other that is played by the first classes' superclasses at any level (on the left).

In our example, Librarian inherits all attributes defined in LibraryUser, and so these are transformed to new DataAIOs placed inside the Librarian view entity, in the Librarian interaction space (see example section at the bottom of fig. 5.5).

5.2.4 DM2UIM04: Transform derived entities and derived attributes

Derived entities, or views, are transformed in a way similar to base entities. A derived entity also generates an interaction space with a view entity block. The difference is on the fact that derived entities only have derived attributes. Only DataAIOs generated from attributes referencing attributes of the target base entity are editable. All other attributes' DataAIOs are read-only. Fig. 5.6 illustrates this transformation rule.

Derived attributes existing in base entities are transformed similarly to at-

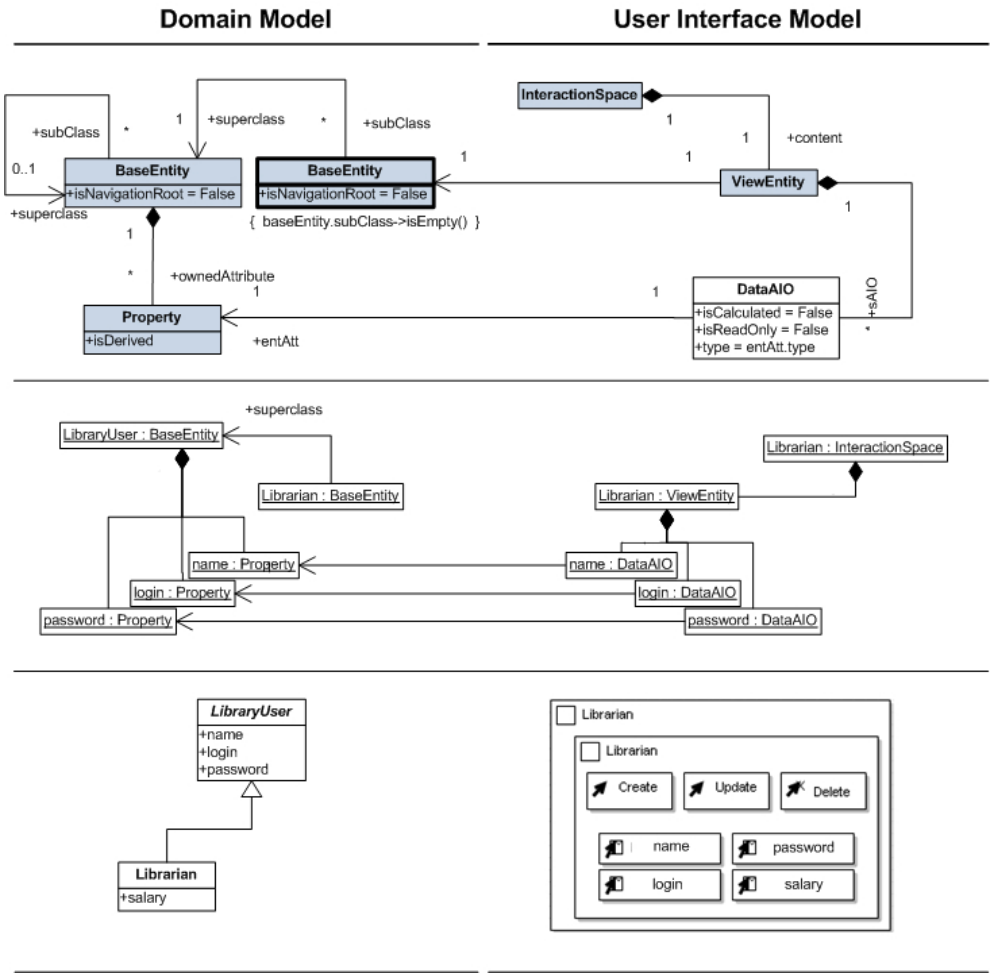

Figure 5.5: DM2UIM03: Adding inherited properties to a leaf class' corresponding view entity, previously generated.

tributes of derived entities. Fig. 5.7 illustrates the transformation rule for derived attributes in base entities. Derived attributes in base entities allways generate read-only DataAIOs.

5.2.5 DM2UIM05: Transform 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 interaction spaces. The elements generated depend on the kind of relationship (composition, aggregation and simple association have slightly different treatment), its multiplicity (to-one and to-many are treated differently), mandatory

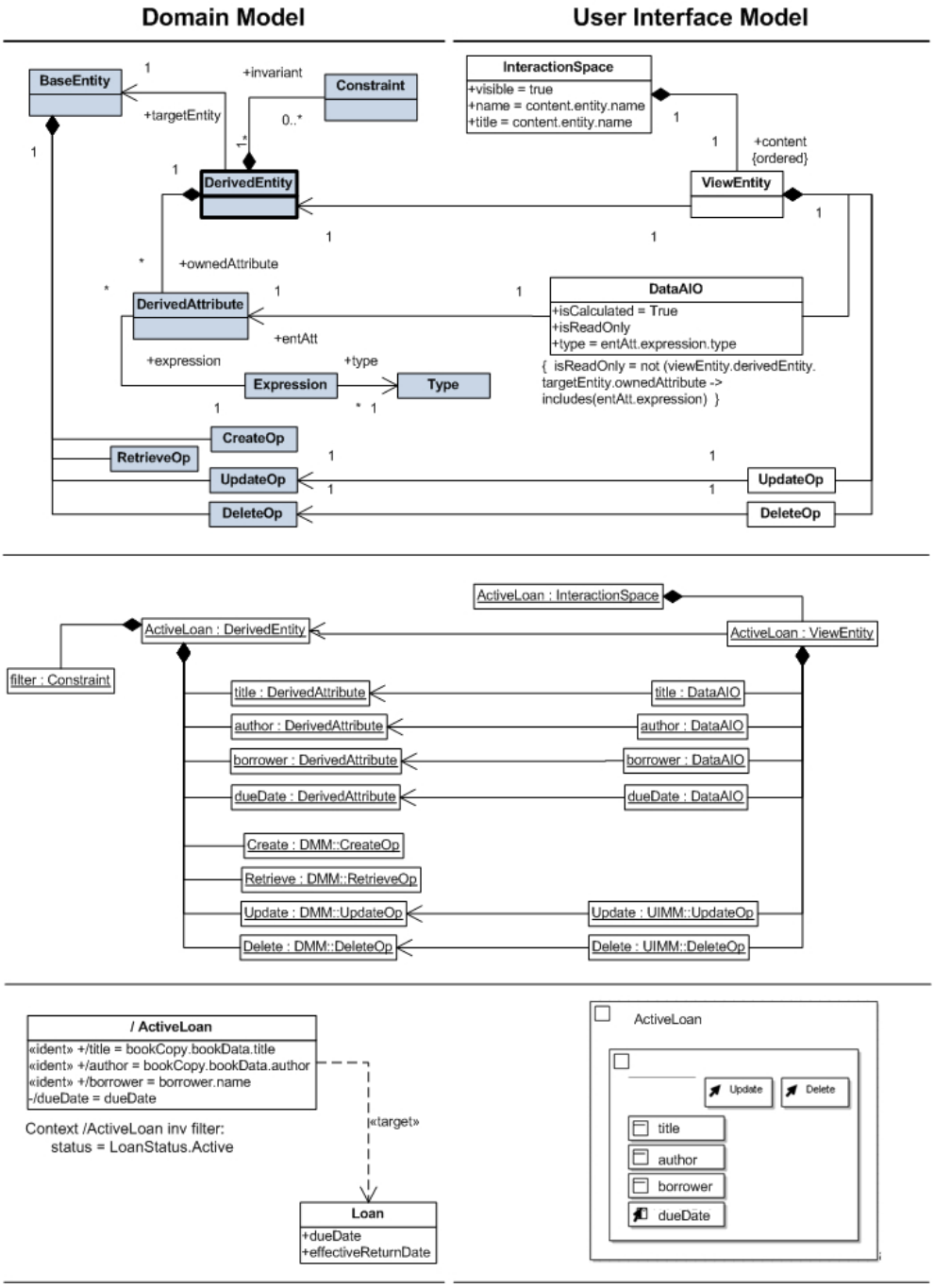

Figure 5.6: DM2UIM04a: Transforming a view, or derived entity, into UIM elements.

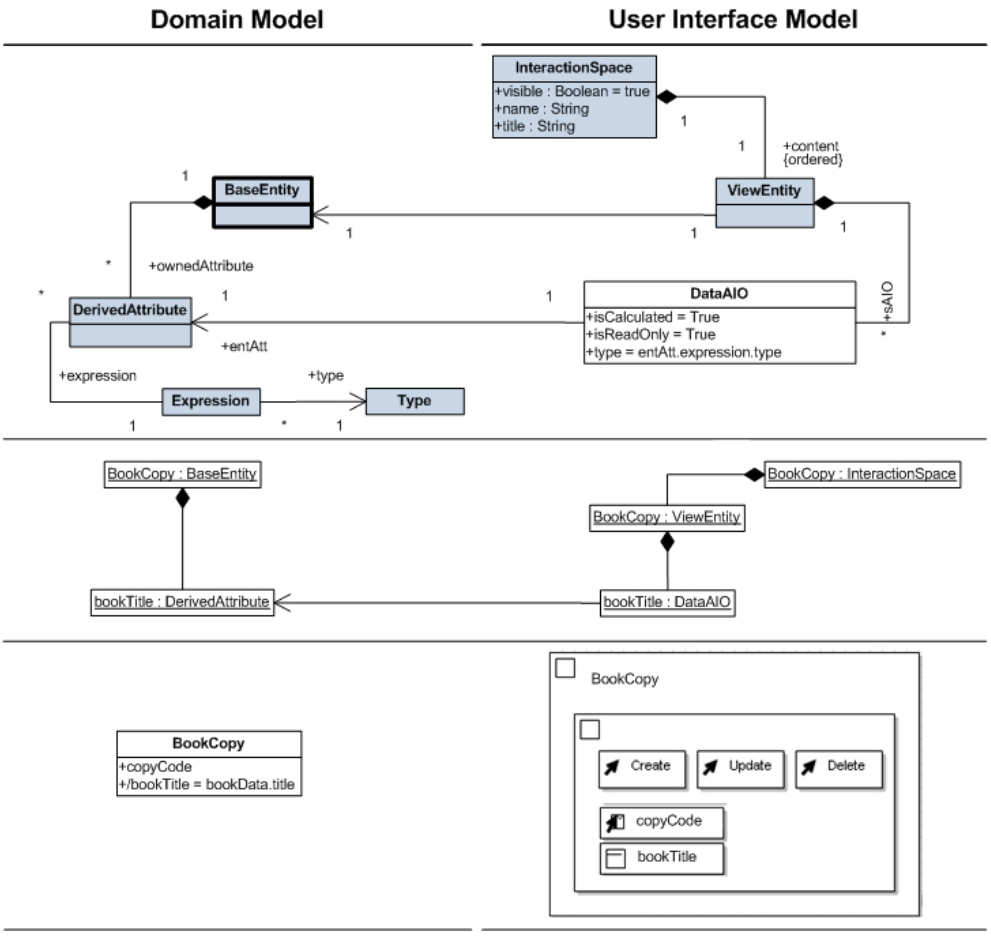

Figure 5.7: DM2UIM04b: Transforming derived attributes of base entities.

ends, and the navigation path followed.

DM2UIM05a: "To-many" dependent relations

Figure 5.8 illustrates the transformation rule for generating UIM elements for a "to-many" dependent relation. The dependence arises from the fact that an entity instance on the other side of the relation is mandatory, meaning that one and only one instance, on the "one" side, may collaborate in this relation (one-to-many relation). The "to-many" side of the one-to-many relation generates a ViewRelatedList block, which is added to the interaction space generated from the entity on the "one" side (the one that is mandatory), and shows the identifying attributes of the related instances of the class in the "many" side (the instances that belong to the collection property at the end of the relationship). From that ViewRelatedList, appropriate ActionAIOs turn possible to create new related instances, edit an existing instance (update or delete) or simply view all attributes

of a related instance.

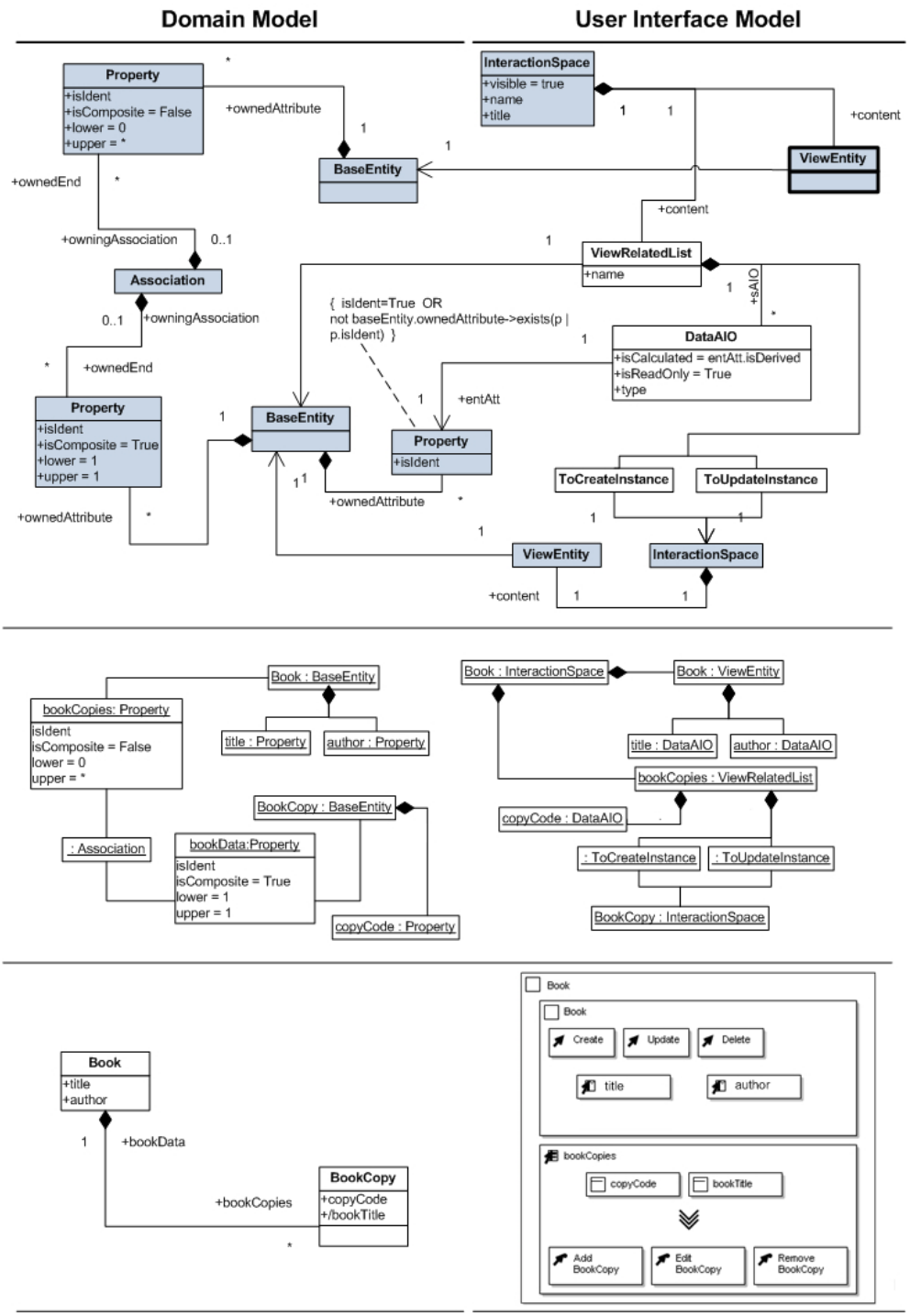

Figure 5.8: DM2UIM05a: Transforming a to-many dependent relation to UI model elements.

The information that is shown about related objects is the value of the identifying attributes (marked with the "ident" stereotype that correspond to the ones having a value of *True* in the isIdent meta-attribute). If no attribute is marked with the "ident" stereotype, then all the attributes are considered identifying attributes. Role names are used to name the interaction block that groups the identifying attributes. If a role name is not provided, it is used the class name, instead.

The *ToCreateInstance* and *ToUpdateInstance* action AIOs that belong to the *ViewRelatedList* being generated must lead to an interaction space with a *ViewEntity* related to the same *BaseEntity* to which *ViewRelatedList* is related. This is assured by the following constraint, which is not in the figure due to the lack of space:

Context ViewRelatedList inv: self.baseEntity = self.toCreateInstance.interactionSpace.baseEntity
and
self.baseEntity = self.toUpdateInstance.interactionSpace.baseEntity

In Fig. 5.9 the UI concrete elements generated from the DM 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 book data identified by title and author, which are identifying attributes ("ident") in class Book.

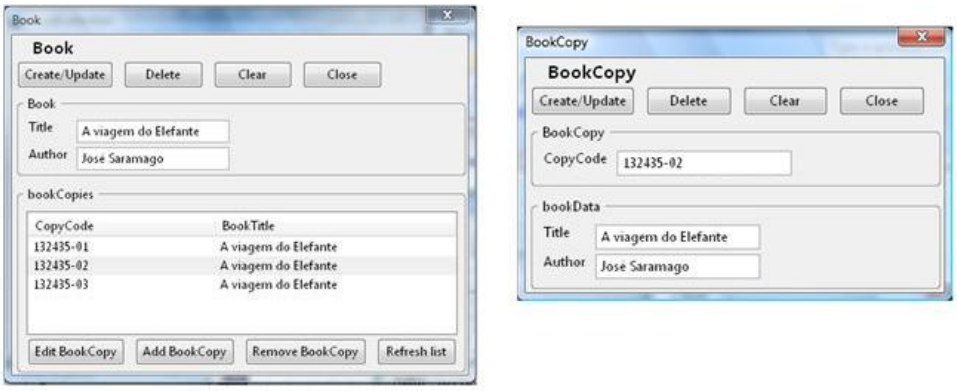

Figure 5.9: Example forms generated from the LibrarySystem DM and generated UIM, illustrating a one-to-many composition association.

In this and the next rules, for keeping the figure simple, some attributes that were already represented in previous figures, are omitted, as they have the same treatment as in the preceding rules.

The "to-one" side of the relation is addressed in subsection "DM2UIM05c: *To-one* relations".

DM2UIM05b: "To-many" independent relations

In the case of an independent "to-many" relationship, where there isn't a one and only restriction on either of its ends, as is the case of the one-to-many association between Shelf and Book, in our example, the UI elements generated are somewhat different. Figure. 5.10 illustrates the transformation rule for generating UIM elements for a "to-many" independent relation. In this case, elements in the "many" side of the relation don't have to be associated to an element in the "one" side of the relation, and so the elements to be linked may already exist without being in this relation. As can be seen in fig. 5.10, a ViewRelatedList block is added to the interaction space corresponding to the entity in side "one", showing the identifying attributes of the related instances of the class in the "to-many" side. That list shows the currently related instances, and allows the navigation to an items selection space for modifying the selection of related instances.

The difference between compositions and aggregations or simple associations is only visible in the value of the $isCollapsable$ meta-attribute in the generated ViewRelatedList block. In the final UI, this means that listing blocks generated for compositions are allways expanded ($isCollapsable = False$), and listing blocks generated for aggregations and associations are, by default, collapsed, but may be expanded by the user ($isCollapsable = True$).

DM2UIM05c: "To-one" relations

In the "to-one" direction of a "one-to-many" relation (Fig. 5.11), a ViewRelatedEntity block is added showing the identifying attributes of the related instance of the class in the "to-one" side. It is also provided a Navigation ActionAIO that leads to an ItemSelectionSpace that enables the selection or the modification of the existing selection of a related instance. This is done no matter if the "one" side is mandatory or not.

In the UI final code, generated by M2C from the UIM and the DM, 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 Fig. 4.9). Fig. 5.12 (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 not previously defined when the user arrives to the Loan form, and both are selectable

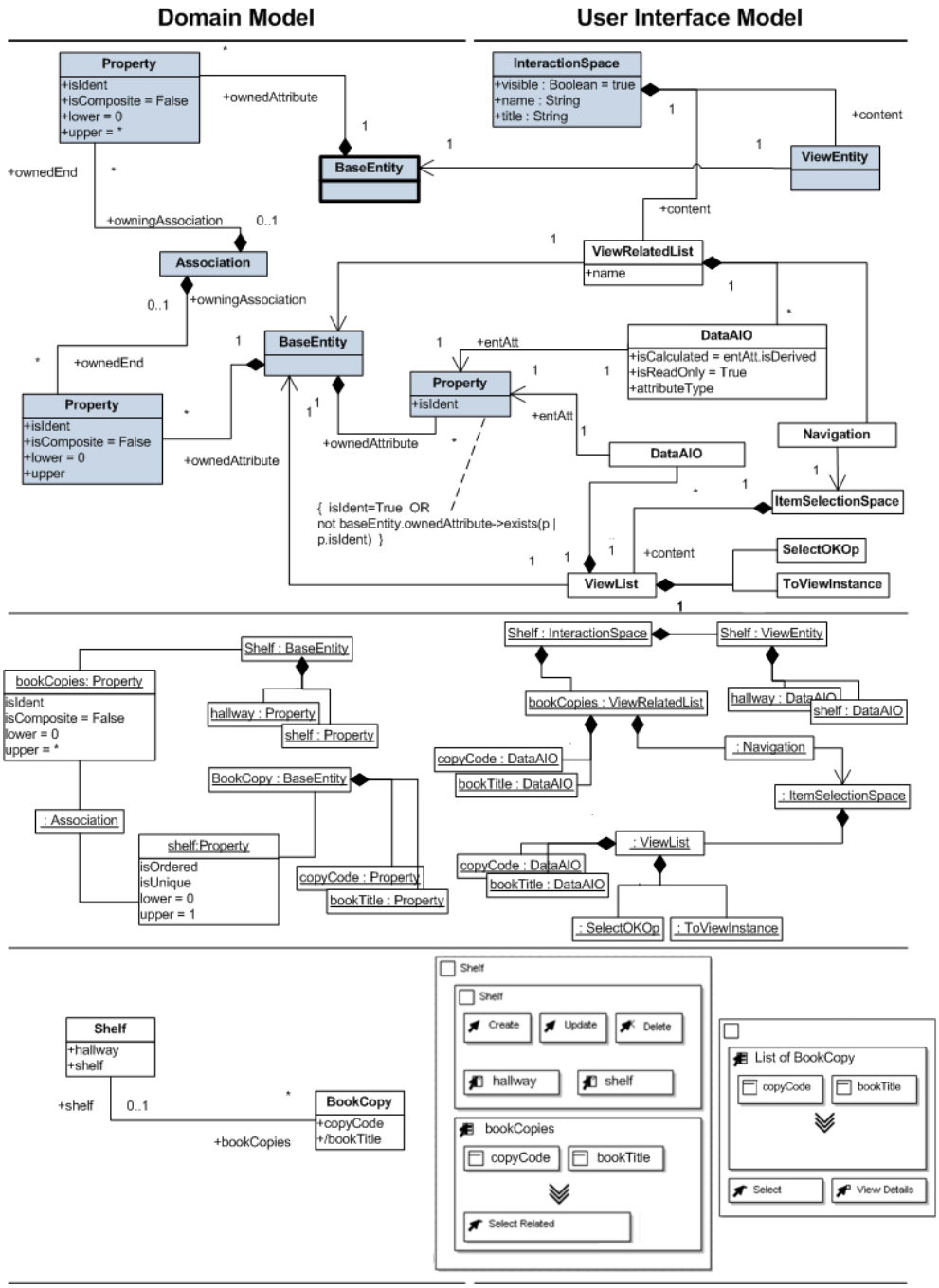

Figure 5.10: DM2UIM05b: Transforming a to-many independent relation to UI model elements.

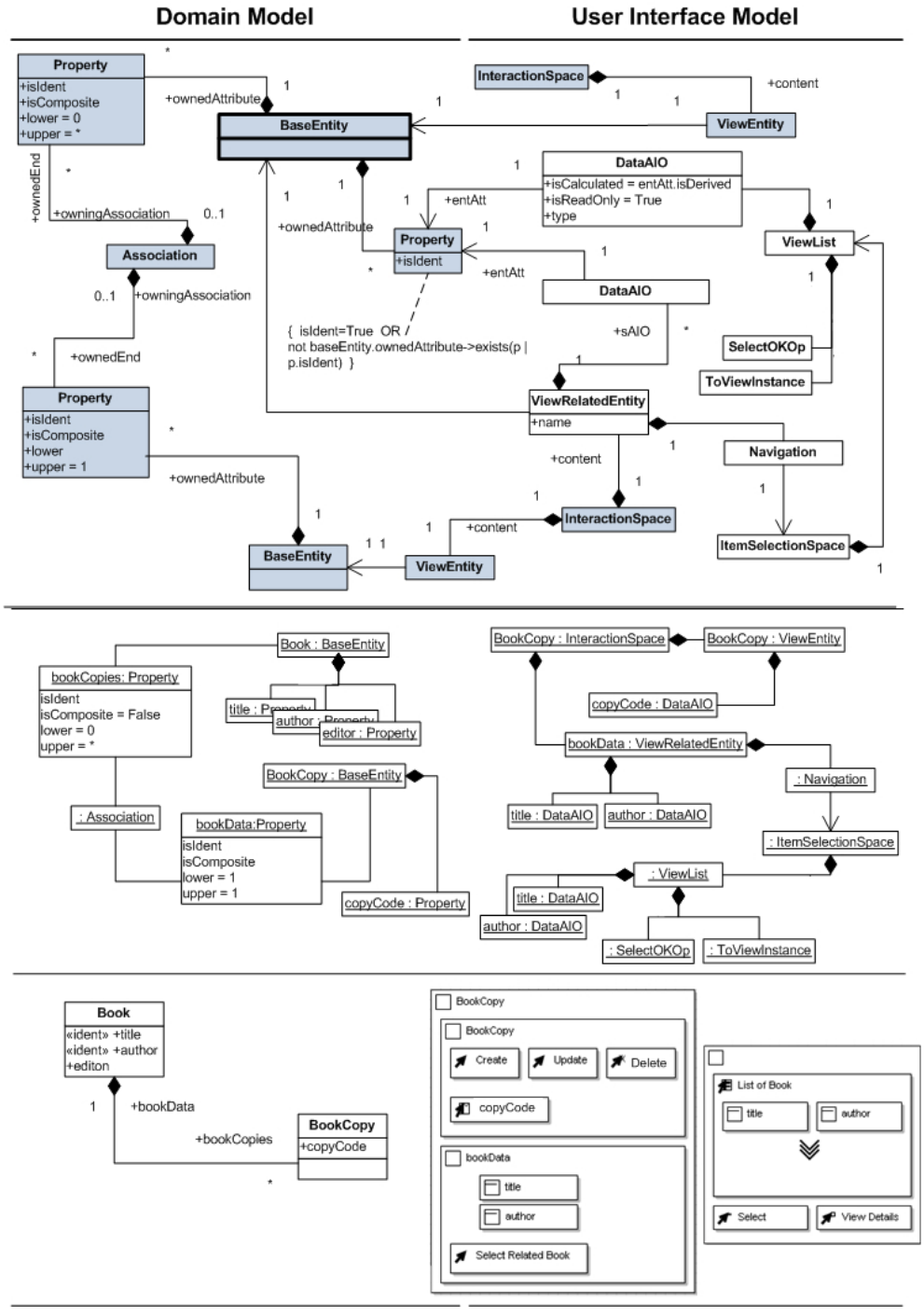

Figure 5.11: DM2UIM05c: Transforming a to-one relation to UI model elements.

from the Loan window. Fig. 5.12 (b) shows the window that appears when navigating from 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.

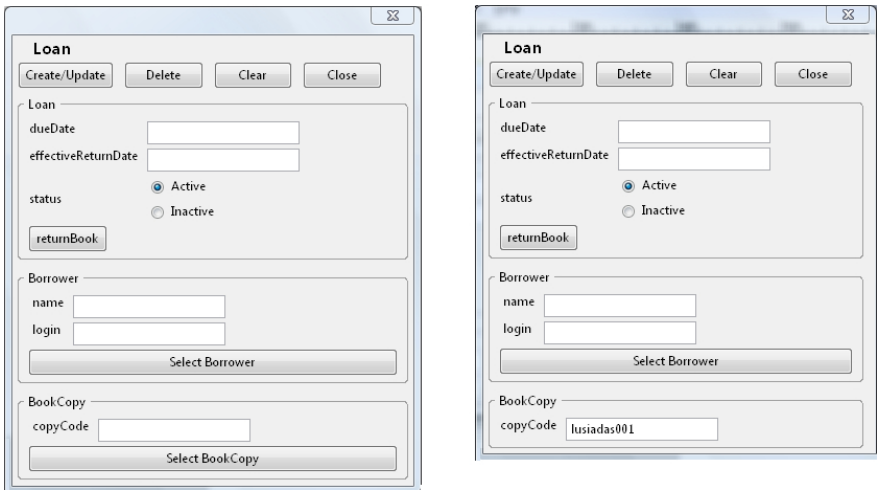

Figure 5.12: (a) Window Loan that is shown when navigating directly to an instance of class Loan. (b) Window Loan, which is shown when navigating from a BookCopy instance to an instance of class Loan.

Many-to-many associations or aggregations

Many-to-many associations or aggregations are treated as two "to-many" independent relations, and so the rule in Fig. 5.10 is applied twice, one in each direction.

5.2.6 DM2UIM06: Handling user defined operations

User defined operations (see rule in Fig. 5.13) generate a Navigation action AIO, which leads to an input parameters space, for accepting the operation's parameters, which in turn, when submitted, calls the domain entity operation, waits for its completion, and then navigates to an operation's output result space that shows the operation's result, if any, and a notification message informing about the success or not of the operation execution.

As before, some attributes are omitted for simplifying the figure.

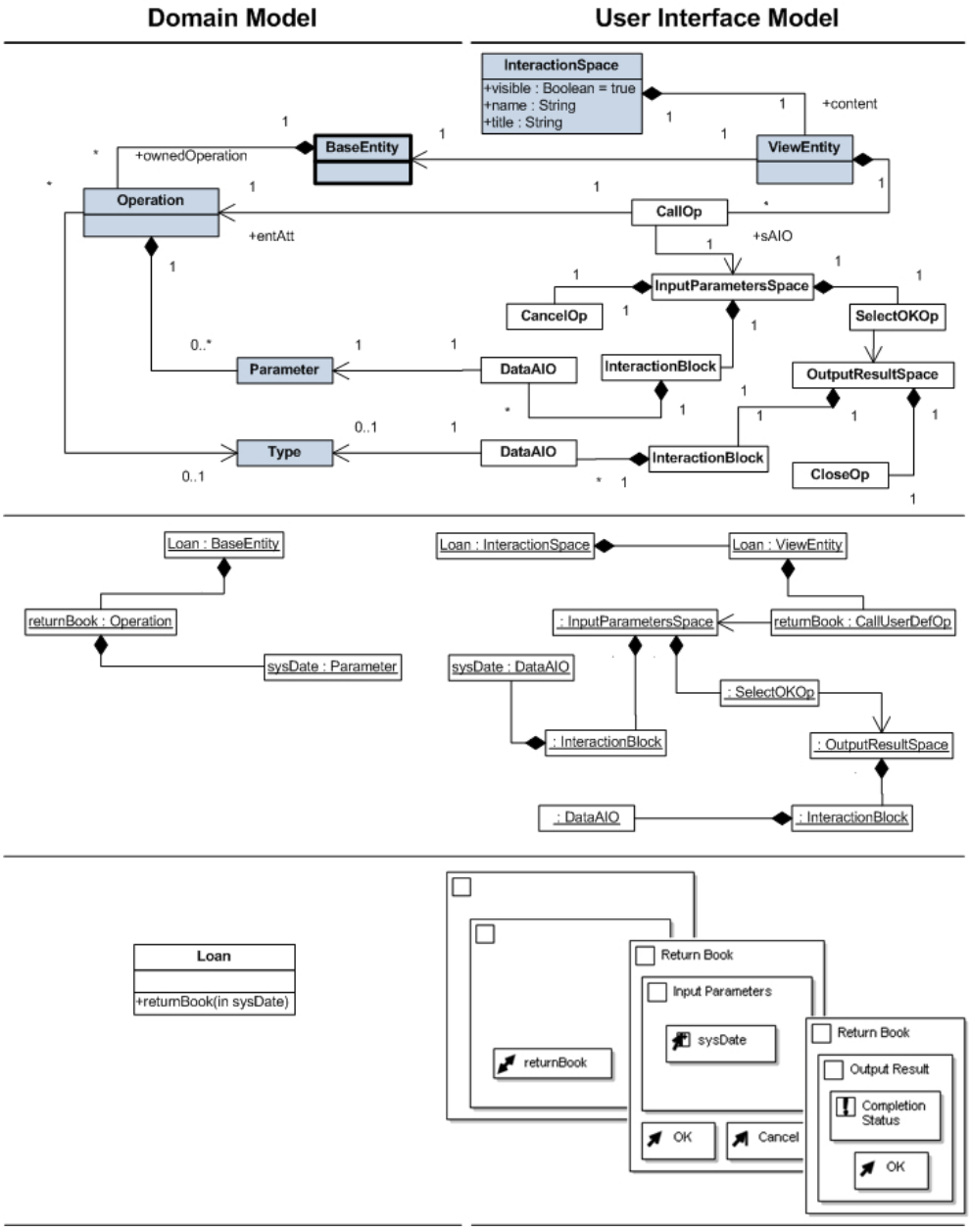

Figure 5.13: DM2UIM06: Transforming an user defined operation to UI model elements.

In the LibrarySystem example, a user defined operation appears in entity Loan, and in fig. 5.13 one can see that two interaction spaces are generated (an InputParametersSpace and an OutputResultSpace) and an ActionAIO is added

to the ViewEntity in the interaction space corresponding to the base entity that has the user defined operation.

5.2.7 DM2UIM07: Transform the navigation root

When the UIM is generated solely from the domain model, which is the case in the model-to-model process DM2UIM, it is needed that one, and only one, class is identified as navigation root. The rule, illustrated in Fig. 5.14, shows what is generated from the navigation root, namely an ActorMainSpace is generated as an entry point in the system's user interface, and each aggregation relation from the root navigation class to other entities gives origin to menu items within a unique menu in the generated ActorMainSpace.

In the LibrarySystem example, this transformation rule generates menu items for BookCollection, BorrowerCollection, LoanCollection, among others.

5.2.8 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. An example of the former is the multiplicity of the attributes, 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.

In the defended approach, constraints are not handled by the UIM generator, but only by the final code generator, when generating the UI prototype from the DM and the UIM. The code generator handles intra- and inter-object constraints by generating data entry validation functions, which are called every time a "Create/Update" button is pressed in the appropriate form. Constraints may be specified, in the domain model, by using an OCL-like abstract language. Constraint expressions may have relational and logical operators, attribute references, constants, etc. Recall subsection 4.4.4 for examples of invariant and precondition OCL constraints.

5.2.9 Handling Domain Triggers

Just like with constraints, and with the algorithmic part of user defined operations, domain triggers are also not handled by the UIM generator, but only by the final code generator. The code generator will inject the trigger actions inside the functions generated for the CRUD operations. The code is injected before, after or instead of the code of the CRUD operation depending on the type of trigger. See subsection 4.4.4 for the example of a trigger using the proposed actions language.

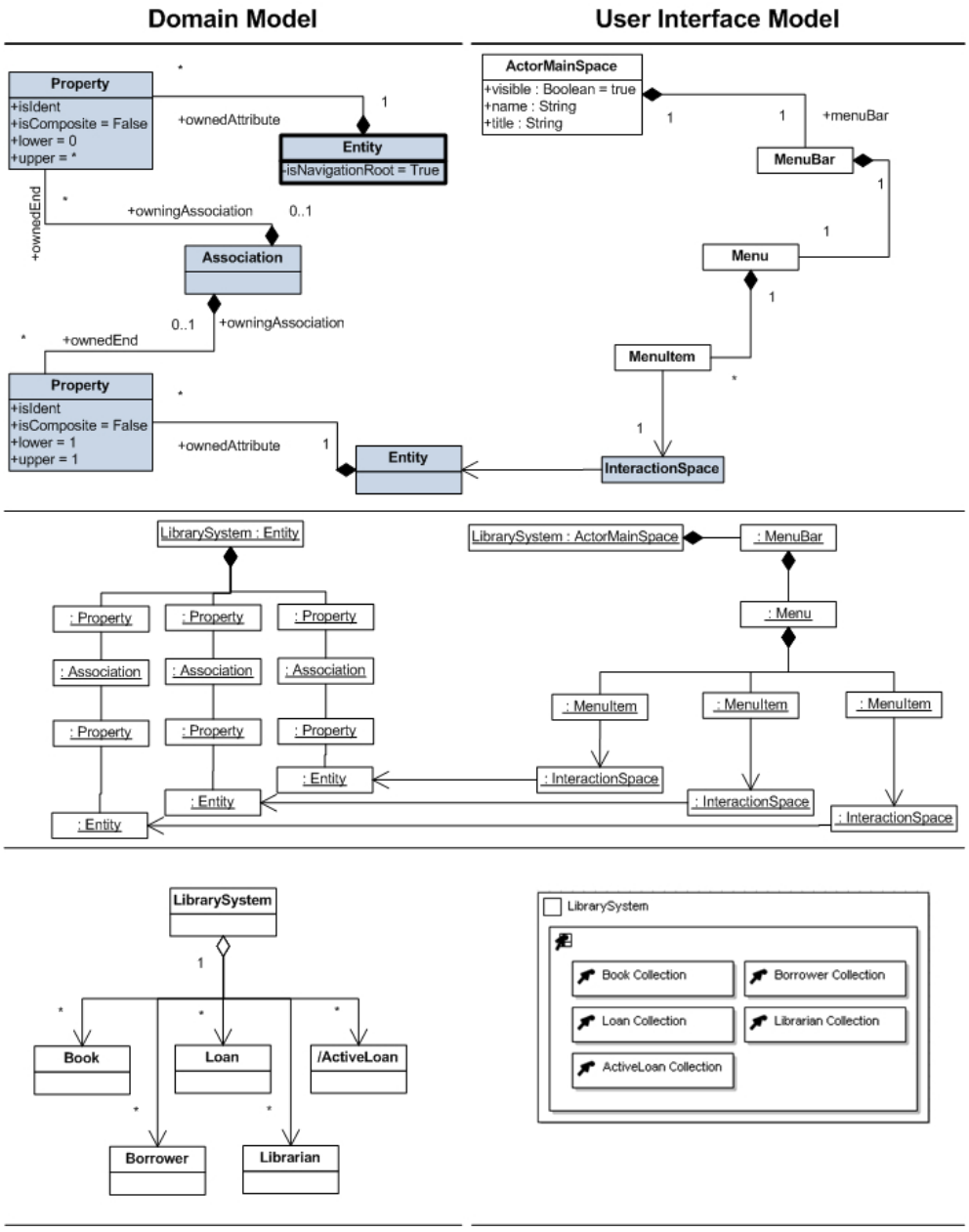

Figure 5.14: DM2UIM07: Transforming the navigation root entity and its relations to other entities into UIM elements.

5.3 Domain and Use Case Integrated Models to User Interface Model Transformation Rules

To better allow the configuration of system functionality and enable its differentiation by actor, the approach presented in this dissertation allows the definition

of a use case model (UCM) in close connection with the domain model. This allows the modeler 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, as addressed in chapter 4.

The meta-attributes, as entity, operation(s), and link type (when needed) associated to each use case, are specified by using tagged-values.

Table 5.2 summarizes the rules for generating UI elements from the UCM and the DM. These were partially previously addressed in [dCF09].

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 5.2: UCM to UIM transformation rules.

A refinement of the Library System example will now be used to illustrate the

transformation rules from a domain model (DM) and a use case model (UCM) to a user interface model/prototype (UIM/UIP). The constructed DM is the same as in the previous section (refer to Fig. 4.9). Such model could have been developed in several iterations; an executable prototype would have been automatically generated and tested at the end of each iteration.

After having a partial or complete DM, the modeler might also develop a UCM. Fig. 4.16 illustrates an extract of a UCM that could have been developed for this system. Table 4.3 shows the entity types and operations associated (via tagged values) with some of the use cases.

The rules for the model-to-model process of DM+UCM2UIM, which is driven by use cases and transforms elements in both the UCM and DM into elements of the UIM, are the following, and its application is made in the presented order:

1. DM+UCM2UIM01: Transform actors, use case packages and links to directly accessible (independent) use cases
2. DM+UCM2UIM02: Transform directly accessible "List Entity" use cases
3. DM+UCM2UIM03: Transform directly accessible "CRUD Entity" (Create) use cases
4. DM+UCM2UIM04: Transform "CRUD Entity use cases" (Create, Retrieve, Update or Delete), accessible through an extension
5. DM+UCM2UIM05: Transform "List Related Entity" use cases, accessible through an inclusion
6. DM+UCM2UIM06: Transform "CRUD related Entity use cases" (Create, Retrieve, Update or Delete), accessible through an extension
7. DM+UCM2UIM07: Transform "Select (one) Related Entity" use cases, accessible through an inclusion
8. DM+UCM2UIM08: Transform Select and Link (several) Related Entity use cases
9. DM+UCM2UIM09: Transform User defined operation use cases
10. DM+UCM2UIM10: Transform Use Case inheritance and specialized use cases, rooted in a directly accessible use case
11. DM+UCM2UIM11: Transform enabling, deactivation and choice relations use cases

For a description of the aforementioned kinds of use cases, refer to section 4.5.2.

5.3.1 DM+UCM2UIM01: Transform actors, use case packages and links to directly accessible (independent) use cases

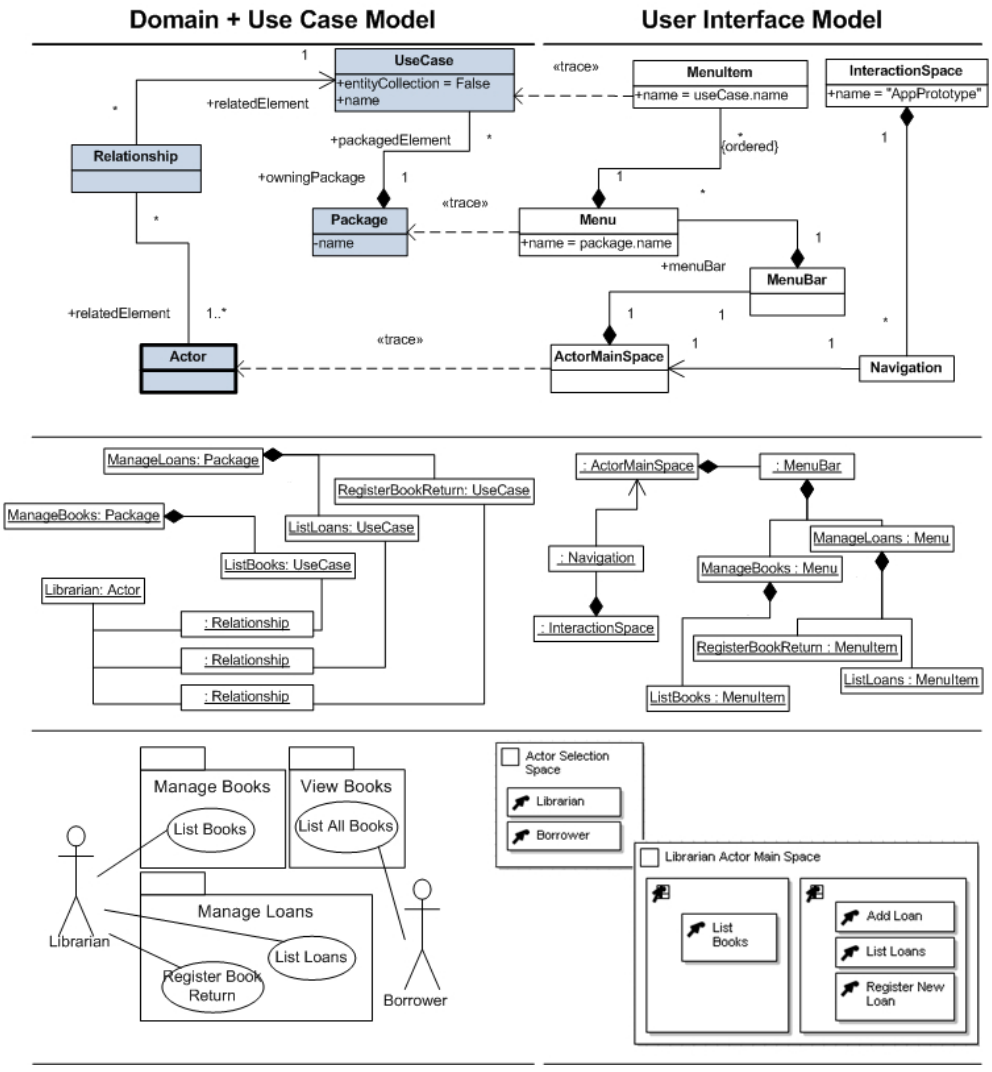

Figure 5.15: DM+UCM2UIM01: Transforming actors, use case packages, and directly accessible use cases.

Each actor originates an ActionAIO element in the application initial InteractionSpace, and an actor's main space, which is accessed through the actor's selection navigation AIO in the initial space (see Fig. 5.15). In our example, after the M2M and M2C processes, 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 gener-

ated 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. 5.15), has menu item "List Books" generated from the directly accessible use case with the same name.

The rule in Fig. 5.15 doesn't generate an interaction space for the directly accessible use cases, as it stops in the menu items that will serve as access points to the interaction spaces that are generated by the application of the following rules.

5.3.2 DM+UCM2UIM02: Transform directly accessible "List Entity" use cases

To be subject to automatic M2M transformation, "List Entity" use cases must be directly accessible from actors. Every use case of type "List Entity" is related to a base or derived entity in the domain model, and for each of these use cases the model transformer generates an interaction space with a ViewList block displaying a full list of existing instances (see Fig. 5.16).

In our example, "List Books" is a List Entity use case from which the "List Books" interaction space has been generated.

The rule in Fig. 5.16 also generates a link from the previously generated menu items to the interaction space, provided that both trace back to the same directly accessible use case.

5.3.3 DM+UCM2UIM03: Transform directly accessible "Create Entity" use cases

Each use case of type "Create entity", which is a use case that targets an entity and a Create operation on that entity, must be directly accessible from an actor and generates an interaction space with a ViewEntity block displaying the attributes' values, with actionAIOs for the CRUD operations allowed (see Fig. 5.17).

As an independent use case, that is a use case for which a context has not been previously set, "CRUD entity" use cases are only able of creating entity instances.

In our example, a "Create entity" use case is, for instance, use case "Add a new Book", which has associated tagged values Entity = "Book" and associatedOp = "Create" (see Table 4.3).

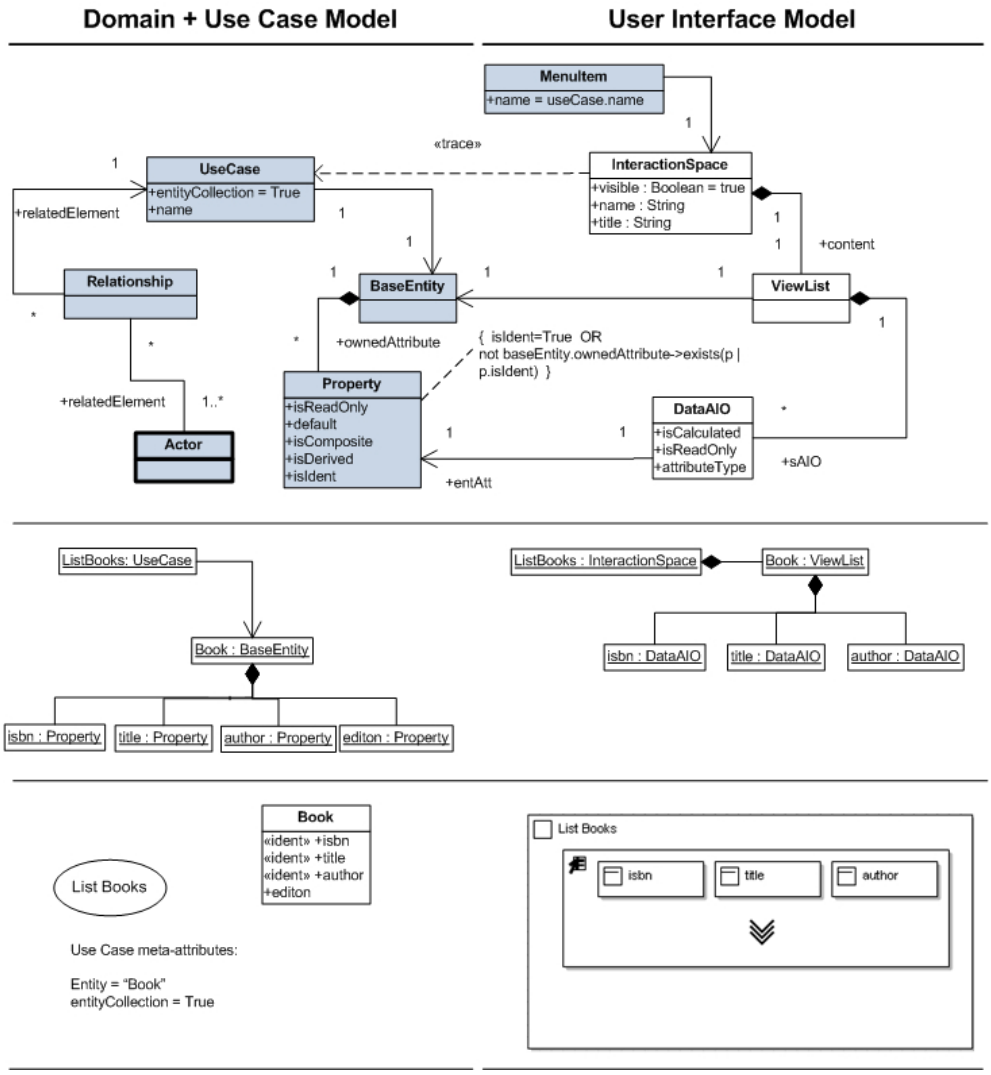

Figure 5.16: DM+UCM2UIM02: Transforming use cases of type "List Entity".

5.3.4 DM+UCM2UIM04: Transform "CRUD Entity" use cases (Create, Retrieve, Update or Delete), accessible through an extension

Each use case of type "List entity" may be extended with a "CRUD Entity" use case, of which a "Create Entity" use case is a special case. "CRUD Entity" use cases target the same entity as the extended "List Entity", and a CRUD operation on that entity. If the operation is other than a Create operation, then the use case is dependent from the context set by the "List entity" use case.

A use case of type "CRUD entity", targeting the same entity as the "List entity" use case it extends, generates an interaction space with a ViewEntity

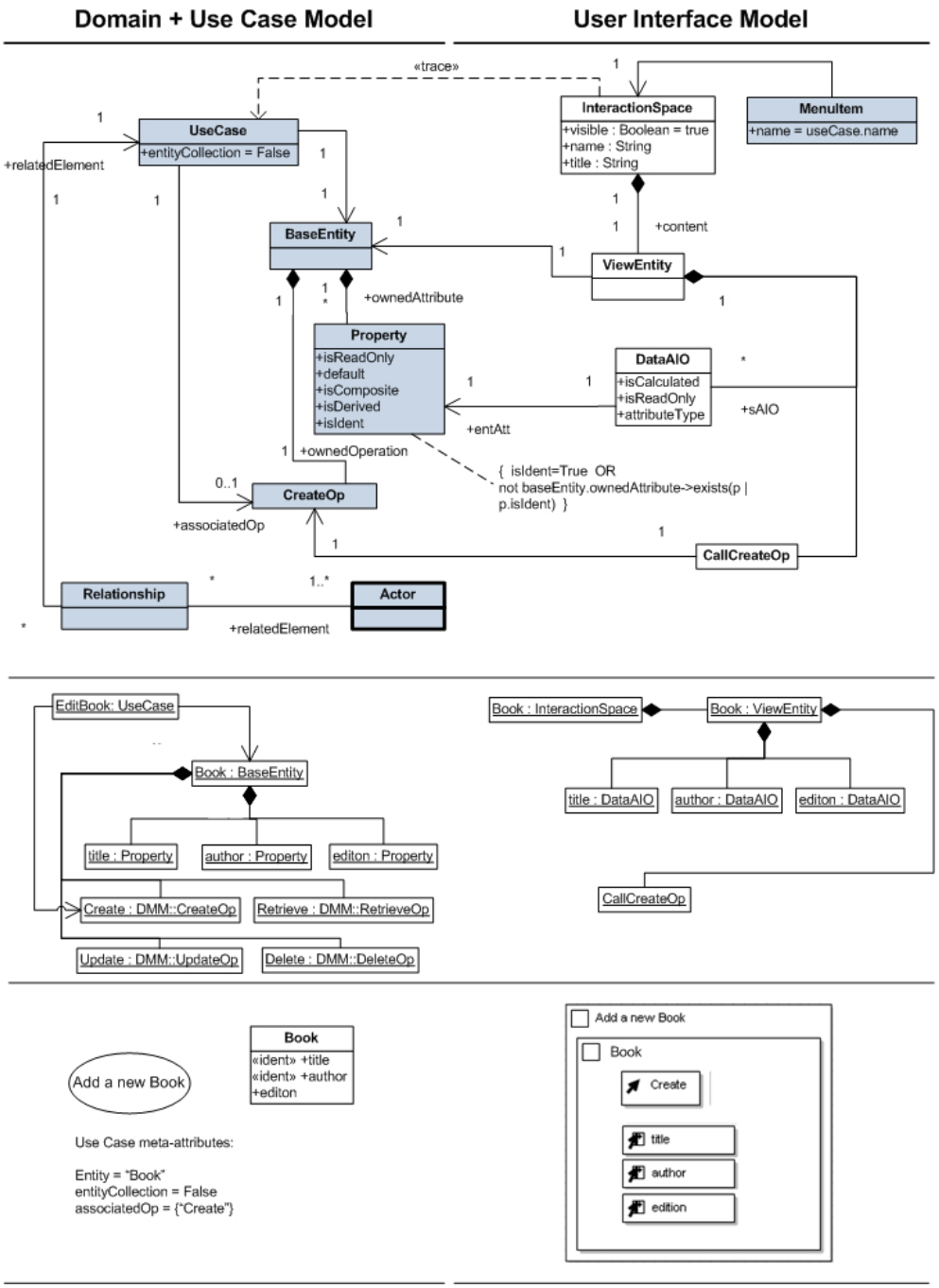

Figure 5.17: DM+UCM2UIM03: Transforming a "Create Entity" use case.

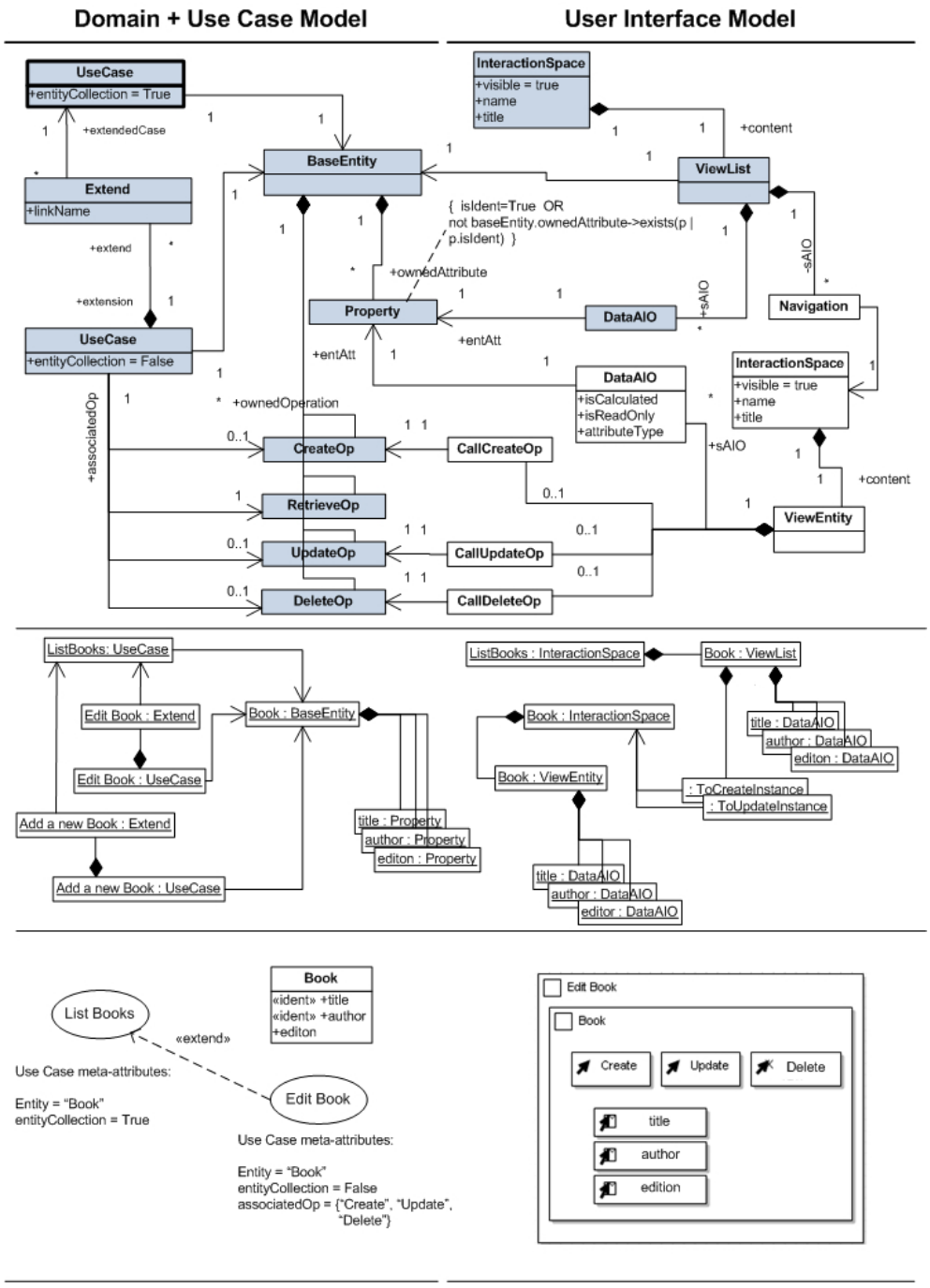

Figure 5.18: DM+UCM2UIM04: Transforming a "CRUD Entity" use case.

block displaying the attributes' values, with actionAIOs for the CRUD operations allowed (see Fig. 5.18).

As a dependent use case, an "Update" or "Delete entity" use case operates on the instance selected in the "List Entity" use case.

In our example, a "CRUD entity" use case is, for instance, use case "Edit Book", which has associated tagged values Entity = "Book" and associatedOp = "Update", "Delete" (recall Table 4.3). "Edit Book" extends use case "List Books".

5.3.5 DM+UCM2UIM05: Transform "List Related Entity" use cases, accessible through an inclusion

Use cases of type "List Related Entity" are a kind of dependent use cases, and thus need that an entity instance is previously set, from which related instances may be listed. For that reason, a relation must exist between the entities of the two use cases related by an inclusion.

Just like with "List Entity" use cases, a "List Related Entity" use case is related to a base or derived entity in the domain model, and the model transformer generates a list displaying the related instances of the target domain model's entity.

Fig. 5.19 shows the rule for transforming a "List Related Entity" use case. In the interaction space of the including use case, the model transformer adds a ViewRelatedList block that lists the instances of the entity associated to the included use case that are related to the instance selected in the including use case.

Use case "List BookCopies", in the LibrarySystem example, included in use case "Edit Book", is, then, an example of a "List Related Entity" use case. In this example, a Book is previously chosen or is created, setting the context for the next list related use case, that is use case "List BookCopies", which, in turn, is responsible for listing the book copies that are related to the book instance that is set in the context, that is the one that is being edited.

5.3.6 DM+UCM2UIM06: Transform "CRUD related Entity" use cases (Create, Retrieve, Update or Delete), accessible through an extension

Fig. 5.20 shows the transformation rule for use cases of type "CRUD related entity". This is another type of dependent use cases, which must have another use case that sets an instance context. This way, a "CRUD related entity" use case must extend a "List Related Entity" use case, and generates an interaction space displaying the attributes values, with ActionAIOs for the CRUD operations allowed. In our example, a CRUD related entity use case is, for instance, use

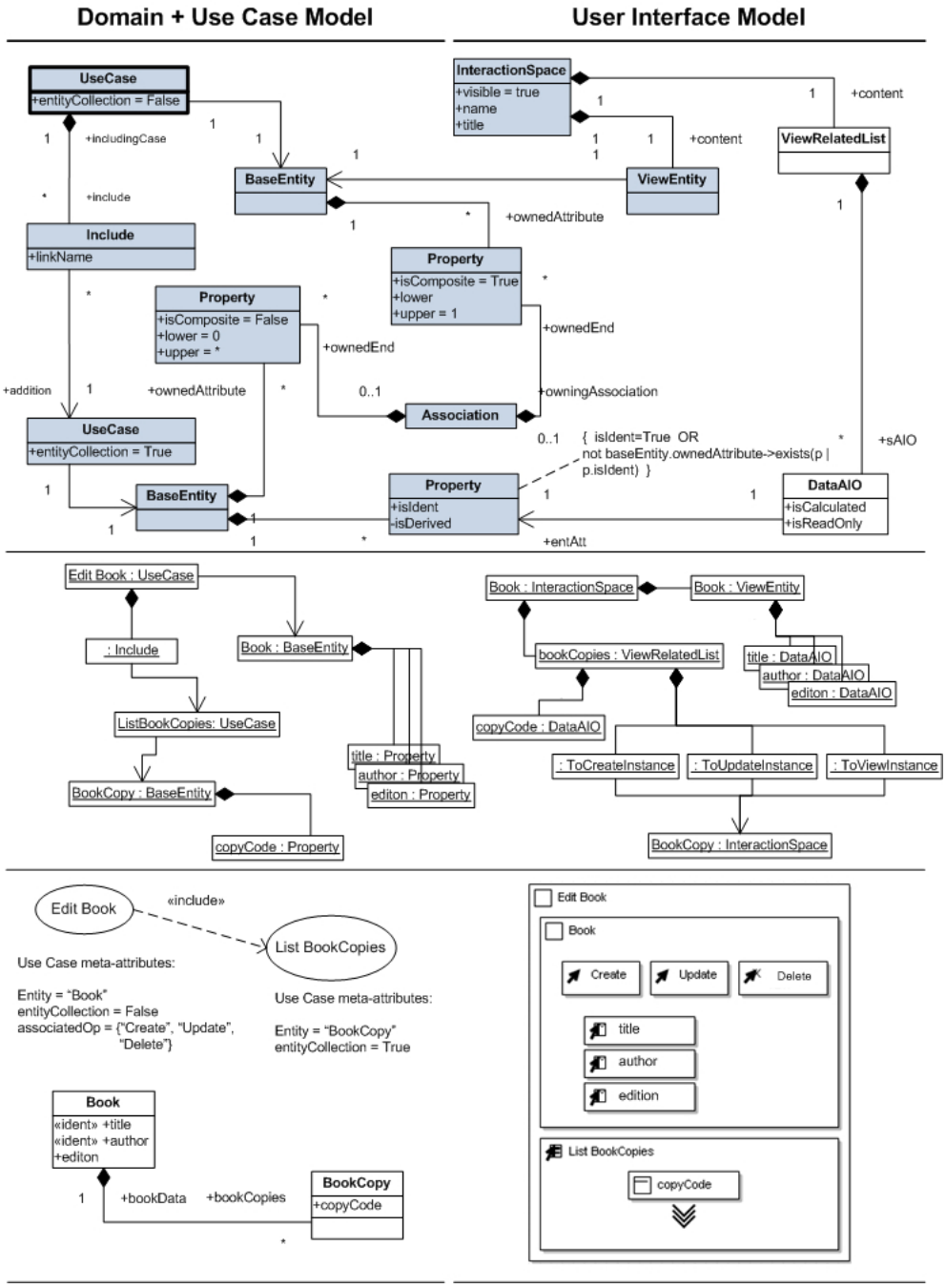

Figure 5.19: DM+UCM2UIM05: Transforming UC inclusion that leads to a "List Related Entity" use case.

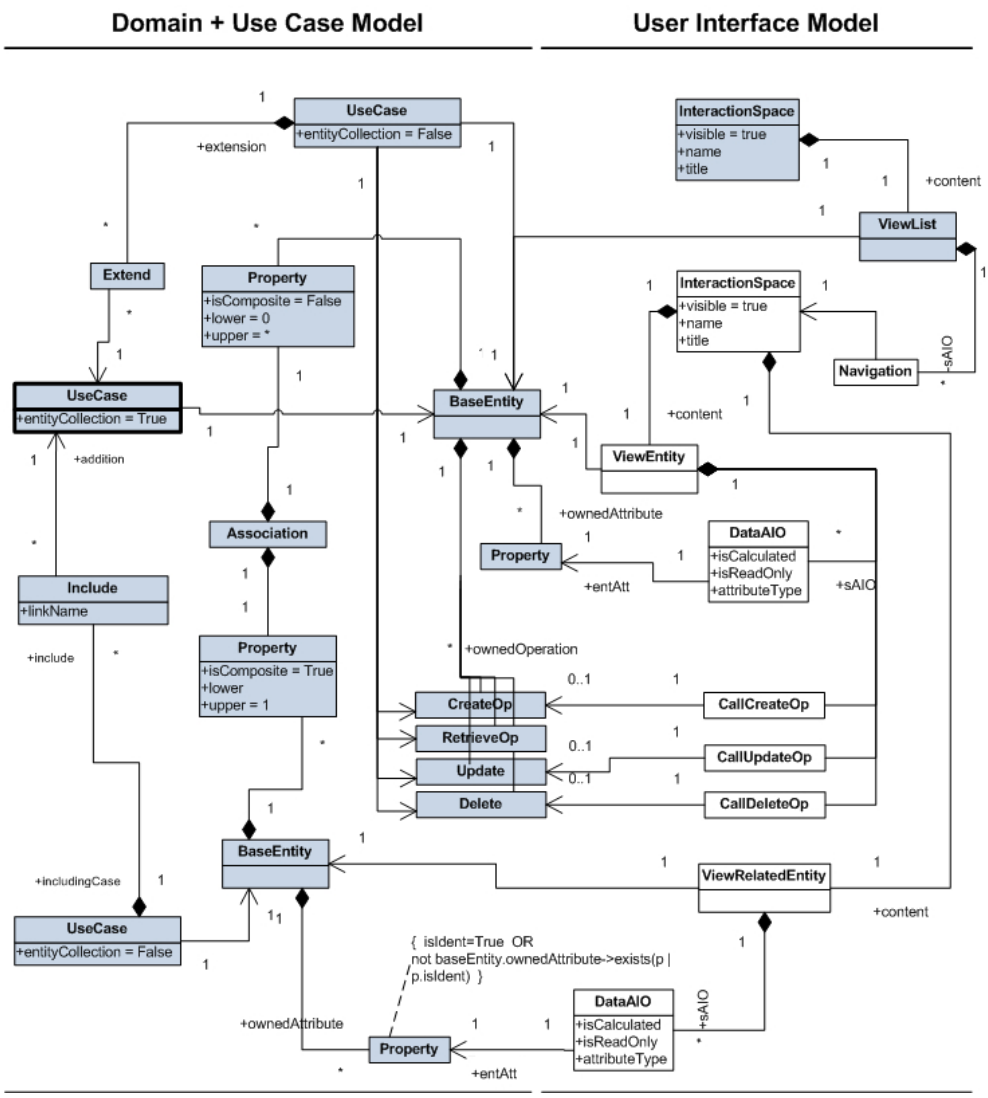

Figure 5.20: DM+UCM2UIM06: Transforming "CRUD Related Entity" use cases.

case "Edit BookCopy", which targets entity "BookCopy" and has associatedOp = "Create", "Update", "Delete" (see Fig. 5.21).

5.3.7 DM+UCM2UIM07: Transform "Select (one) Related Entity" use cases, accessible through an inclusion

Use cases of type "Select Related Entity" are another kind of dependent use cases. "Select Related Entity" use cases also need that an entity instance is

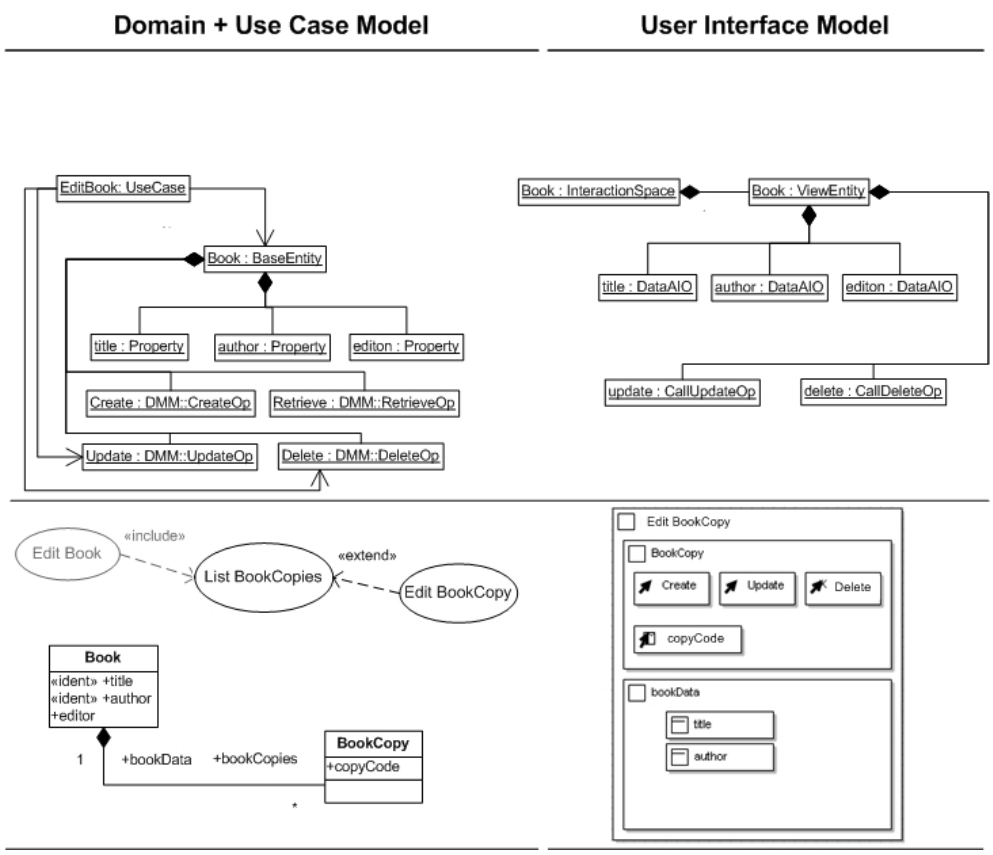

Figure 5.21: Example of "CRUD Related Entity" use case transformation (DM+UCM2UIM06).

previously set, from which a related instance may be accessed. For that reason, in the domain model, a "to-one" relation must exist between the entities of the two use cases related by an inclusion.

Fig. 5.22 shows the rule for transforming a "Select Related Entity" use case. In the interaction space of the including use case, the model transformer adds a ViewRelatedEntity block that shows the identifying attributes of the instance related to the one in the including use case.

From the ViewRelatedEntity, a navigation action AIO leads to an ItemSelectionSpace where a list of all instances of the related entity are shown, and from where one instance can be selected to be linked to the context instance set in the including use case.

In the LibrarySystem example "Select BookCopy" and "Select Borrower" are use cases of type "Select Related Entity", where an independent instance of BookCopy or Borrower, respectively, must be associated to an instance of Loan.

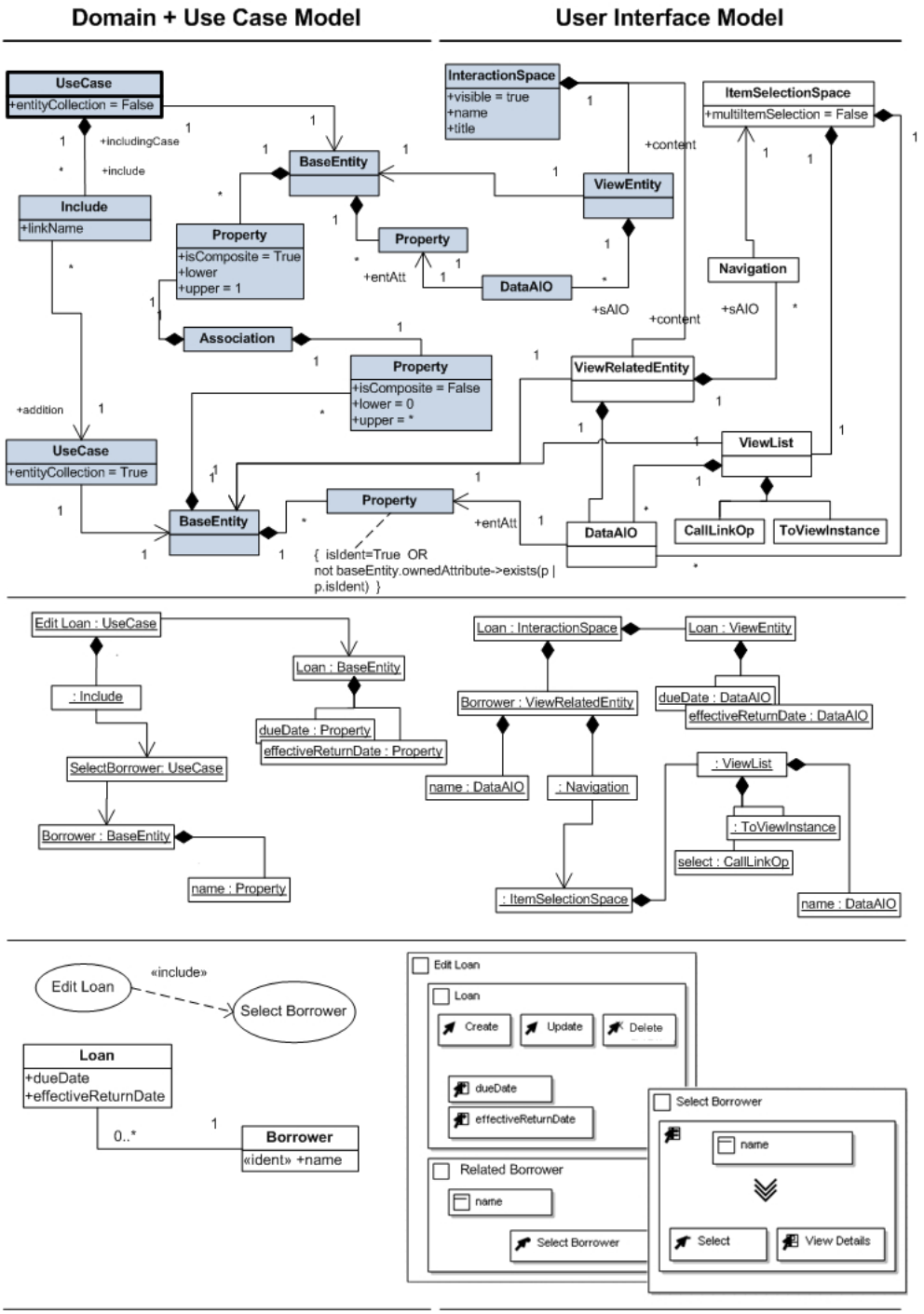

Figure 5.22: DM+UCM2UIM07: Transforming a use case of type "Select Related Entity".

5.3.8 DM+UCM2UIM08: Transform "Select and Link (several) Related Entity" use cases

"Select and Link Related Entity" use cases are also dependent use cases, and so need that an entity instance is previously set, from which related instances may be accessed. For that reason, an independent "to-many" relation must exist between the entities of the two use cases related by an inclusion (recall Fig. 4.12 for the possible types of use cases for different domain model entities relationships).

Fig. 5.23 shows the rule for transforming a "Select and Link Related Entity" use case. In the interaction space of the including use case, the model transformer adds a ViewRelatedList block that shows the identifying attributes of the instances related to the one that set the context in the including use case.

From the ViewRelatedList, a navigation action AIO leads to an ItemSelectionSpace where a list of all instances of the related entity are shown, and from where several can be selected to be linked to the context instance set in the including use case.

Another action AIO, in the included ViewRelatedList, allows for the unlinking of selected related instances.

5.3.9 DM+UCM2UIM09: Transform User defined operation use cases

A "Call User Defined Operation" use case generates a CallUserDefOp action AIO, in the previously generated Interaction space corresponding to the entity where the operation is defined, and an InputParametersSpace, for entering parameters, and an OutputResultSpace, for showing the operation's result, if one exists (see Fig. 5.24).

In our example, this situation appears in Loan. Class Loan defines operation returnBook, with an input parameter, that is ultimately 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 generated with no results, only to inform that the operation is complete. When the operation returns, the entity form is refreshed to be able to show data modified by the operation in the instance's state.

5.3.10 DM+UCM2UIM10: Transform Use Case inheritance and specialized use cases, rooted in a directly accessible use case

Use case inheritance is subject to automatic M2M transformation in the scope of the DM+UCM2UIM process, only if it appears in directly accessible use cases.

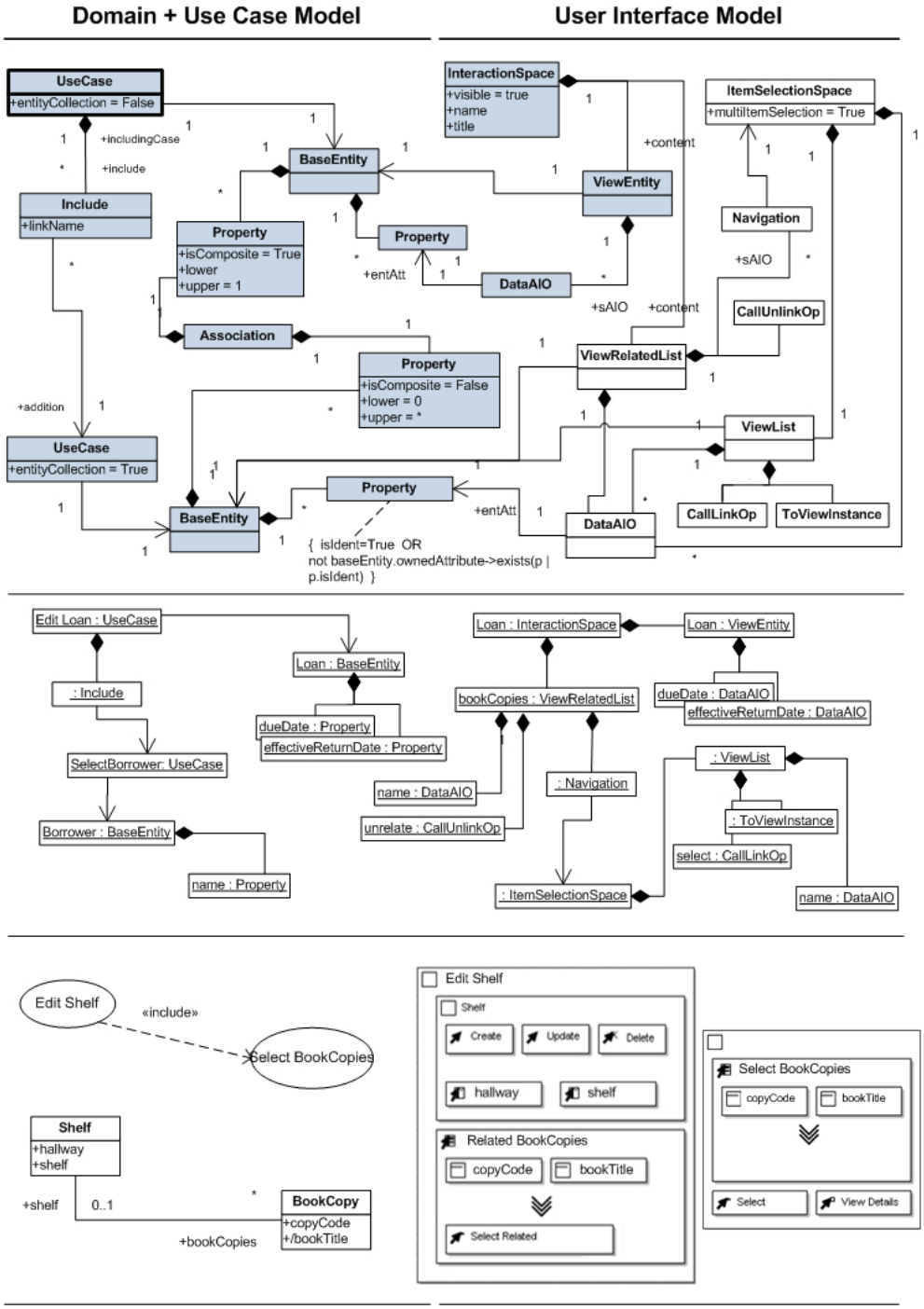

Figure 5.23: DM+UCM2UIM08: Transforming a use case of type "Select and Link Related Entity".

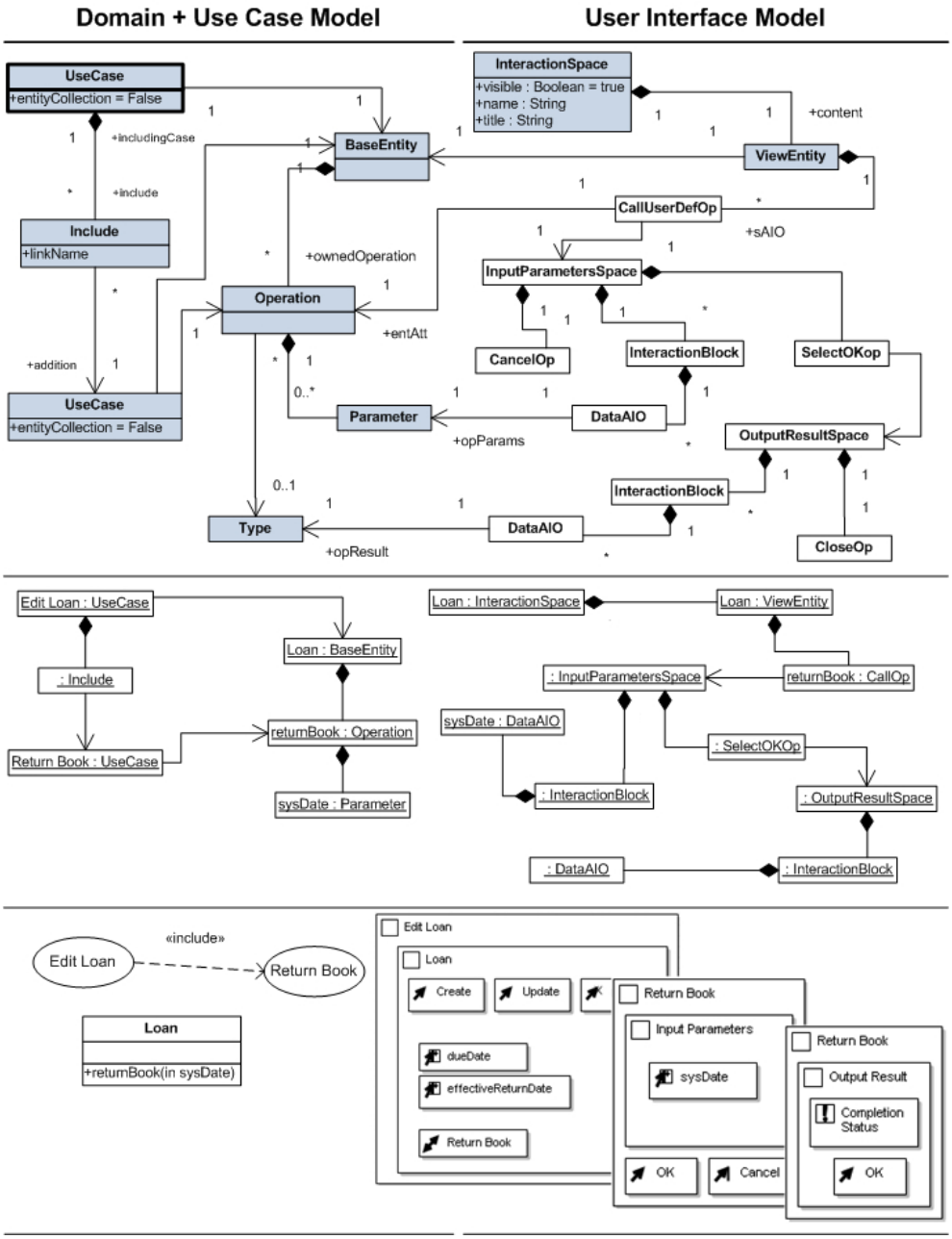

Figure 5.24: DM+UCM2UIM09: Transforming a use case of type "Call User Defined Operation".

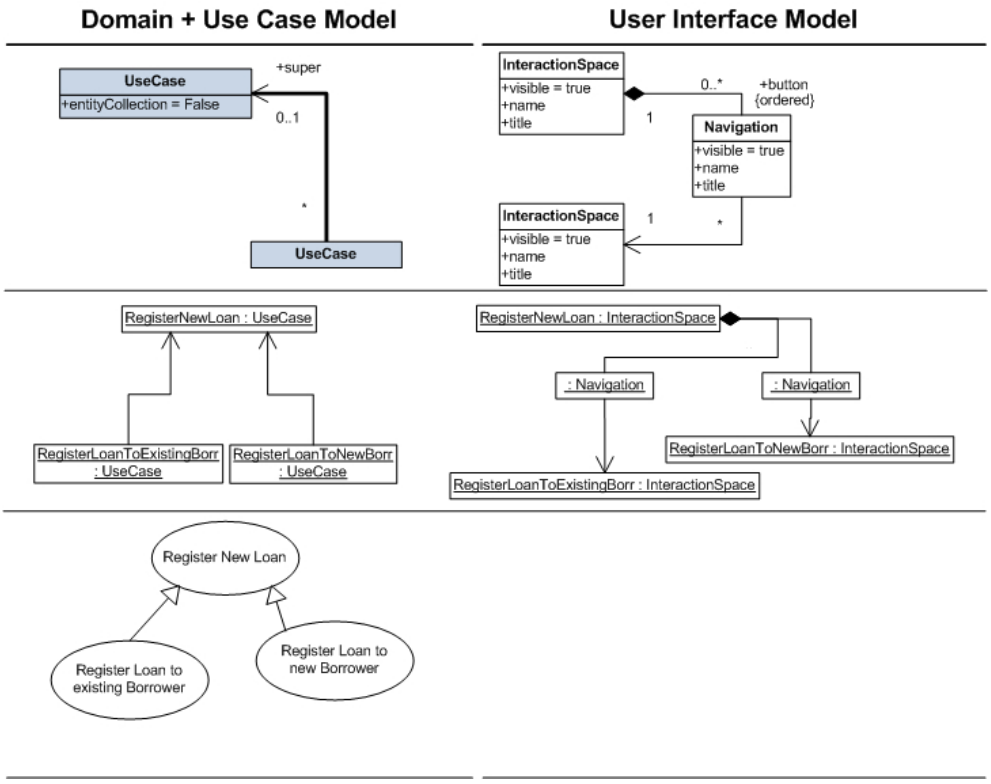

Figure 5.25: DM+UCM2UIM10: Transforming use case inheritance.

In this case, navigation AIOs are generated for accessing the interaction spaces of the inheriting use cases, or sub-use-cases. In this sense, sub-use-cases are treated as if they were extensions of the super-use-case. But, there is indeed a different treatement, as sub-use-cases inherit all inclusions and extensions of the super-use-case, and also all meta-attributes, namelly the ones that provide an integration with the DM. This way, each sub-use-case originates a different interaction space, which is then treated as any other use case. This implies that for each sub-use-case, the previously addressed transformation rules are applied, and if a pattern matching occurs the corresponding transformation takes place.

5.3.11 DM+UCM2UIM11: Transform enabling, deactivation and choice relations use cases

Figure 5.26 shows the transformation rule for an enable relation. This rule applies to enable relations between use cases that are included in one, and the same, use case. In this context, when the enabling use case ends, the second one must be enabled. This is information usefull for the M2C process, when it is generating final code. In the UIM, the relevant information is registered by relating the

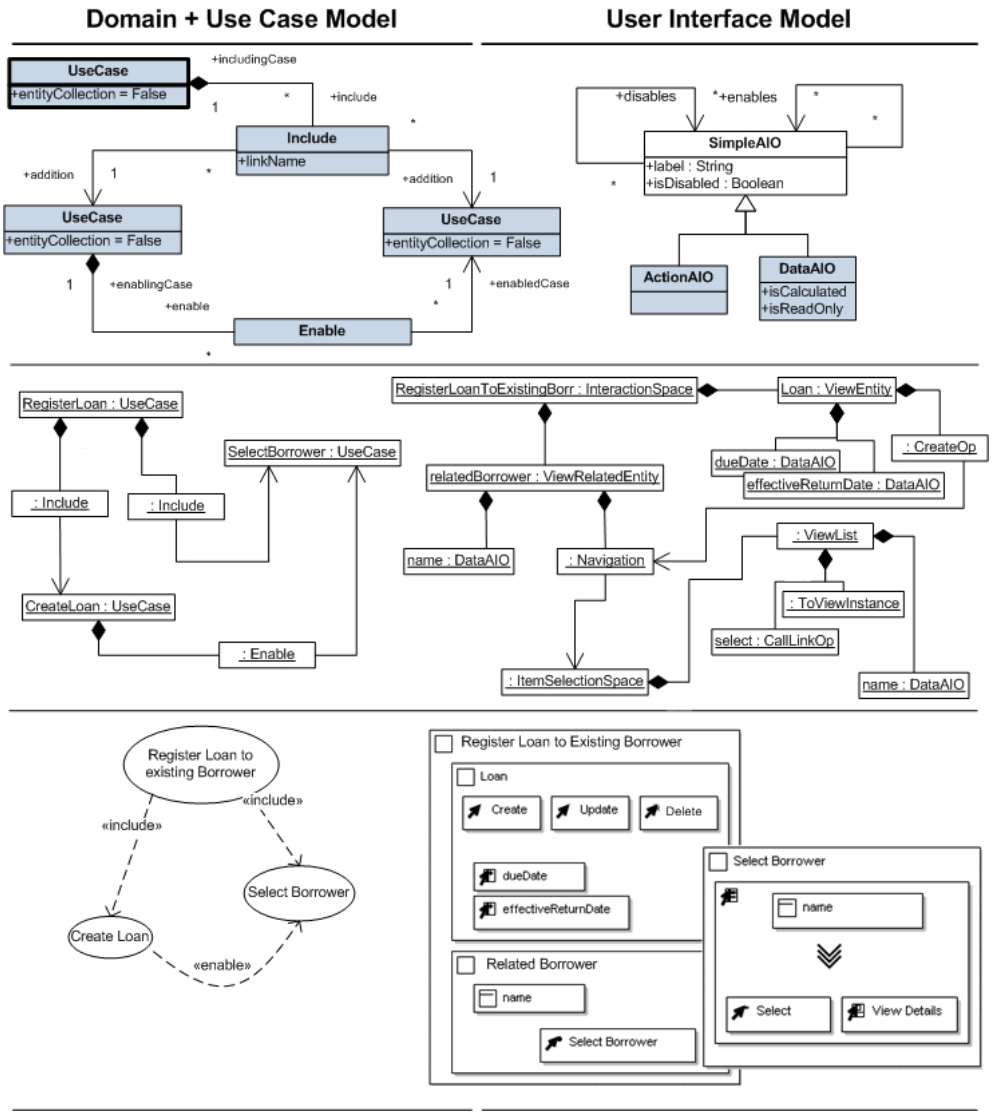

Figure 5.26: DM+UCM2UIM11: Transforming use case enabling to UIM.

SimpleAIO elements involved, through relation "enables".

Similarly, a deactivation relation in the UCM originates a "disables" relation between simpleAIOs in the UIM.

With regard to choice relations, if they involve only two use cases, each one is related to the other through a "disables" relation. If the choice is between several use cases (see Fig. 4.14), they are treated similarly, two at a time.

5.4 Default Use Case Model generation from Domain Model

As stated before, and according to the proposed approach (refer to chapter 4) a default UCM may be derived from the DM 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.

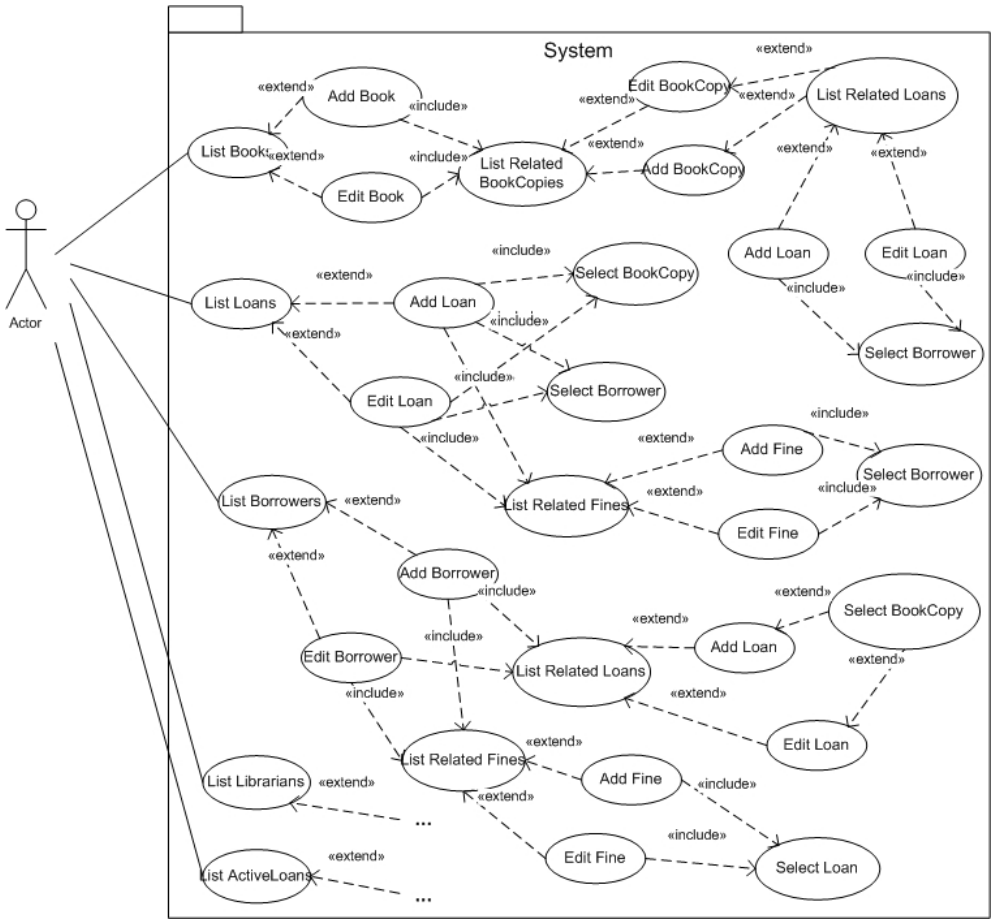

Figure 5.27: Partial default use case model generated from the DM in Fig. 4.9.

The transformation rule execution mechanism works the same way as for DM2UIM and DM+UCM2UIM model-transformation processes. When a pattern matching occurs, on the LHS, the elements of the RHS are generated or modified according to the specified relationships to elements in the source model. The LHS is the source model, which, in this case, is formed by a domain model instance

and, eventualy, by previously generated use case model elements.

This way, the rules for the M2M process DM2UCM, which transforms elements in the DM into elements in the UCM, are the following, and its application follows the presented order:

1. DM2UCM01: Transform root navigation entity and its aggregation relations to other entities
2. DM2UCM02: Transform directly accessible base entities to CRUD UCs
3. DM2UCM03: Transform directly accessible derived entities to CRUD UCs
4. DM2UCM04: Transform "to-many" relations from dependent instances to CRUD related UCs
5. DM2UCM05: Transform "to-many" relations from independent instances to Select Related UCs
6. DM2UCM06: Transform "to-one" relations from dependent or independent instances to Select Related UCs
7. DM2UCM07: Transform user defined operations

5.4.1 DM2UCM01: Transform root navigation entity and its aggregation relations to other entities

Starting from the navigation root entity (class with the isNavigationRoot meta-attribute set to True) an actor is created, linking to List Entity use cases, one for each base or derived entity with an aggregation from the navigation root. Fig. 5.27 partially shows the use case model that is generated by the DM2UCM model-to-model transformation process, from the domain model shown in figure 4.9. Figure 5.28 illustrates this transformation rule.

5.4.2 DM2UCM02: Transform directly accessible base entities to CRUD UCs

Each List Base Entity use case, already generated, shall have extensions for CRUD Entity use cases (Add and Edit). In Fig. 5.27, see, for example, use case "List Books" that links to the only actor and is extended by "Add Book" and "Edit Book". Figure 5.29 shows this transformation rule.

5.4.3 DM2UCM03: Transform directly accessible derived entities to CRUD UCs

Also, each List Derived Entity use case, already generated, shall have extensions for CRUD (Edit) Entity use cases (see rule in Fig. 5.30).

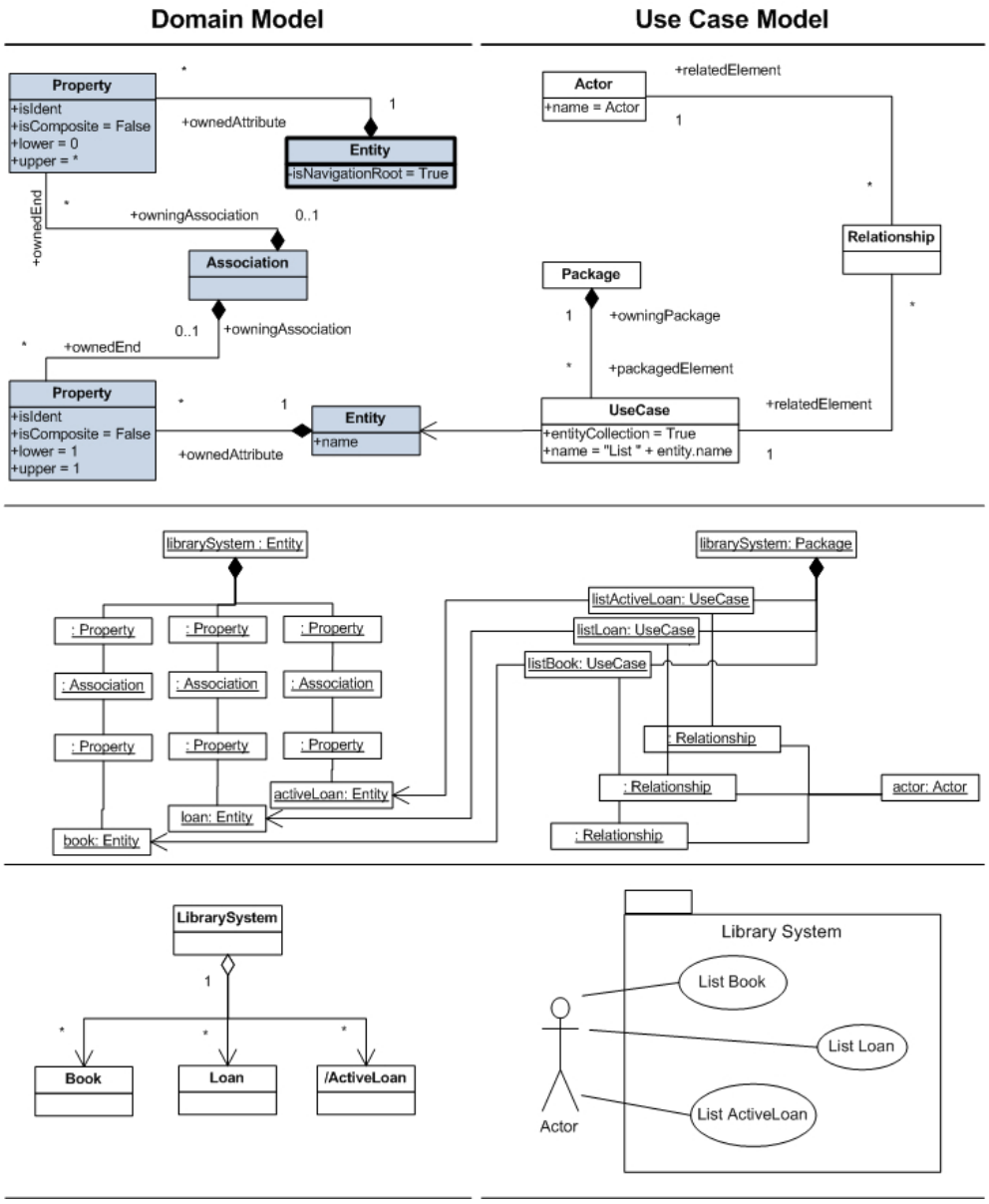

Figure 5.28: DM2UCM01: Transforming root navigation entity and its aggregation relations to other entities to UCM elements.

5.4.4 DM2UCM04: Transform "to-many" relations from dependent instances to CRUD related UCs

A CRUD Entity use case shall include use cases that list related entity instances. Fig. 5.31 shows the rule for generating CRUD related use cases from "to-many" relations having a dependent instance, that is "one-to-many" relations where the entity at the one side is one and only one.

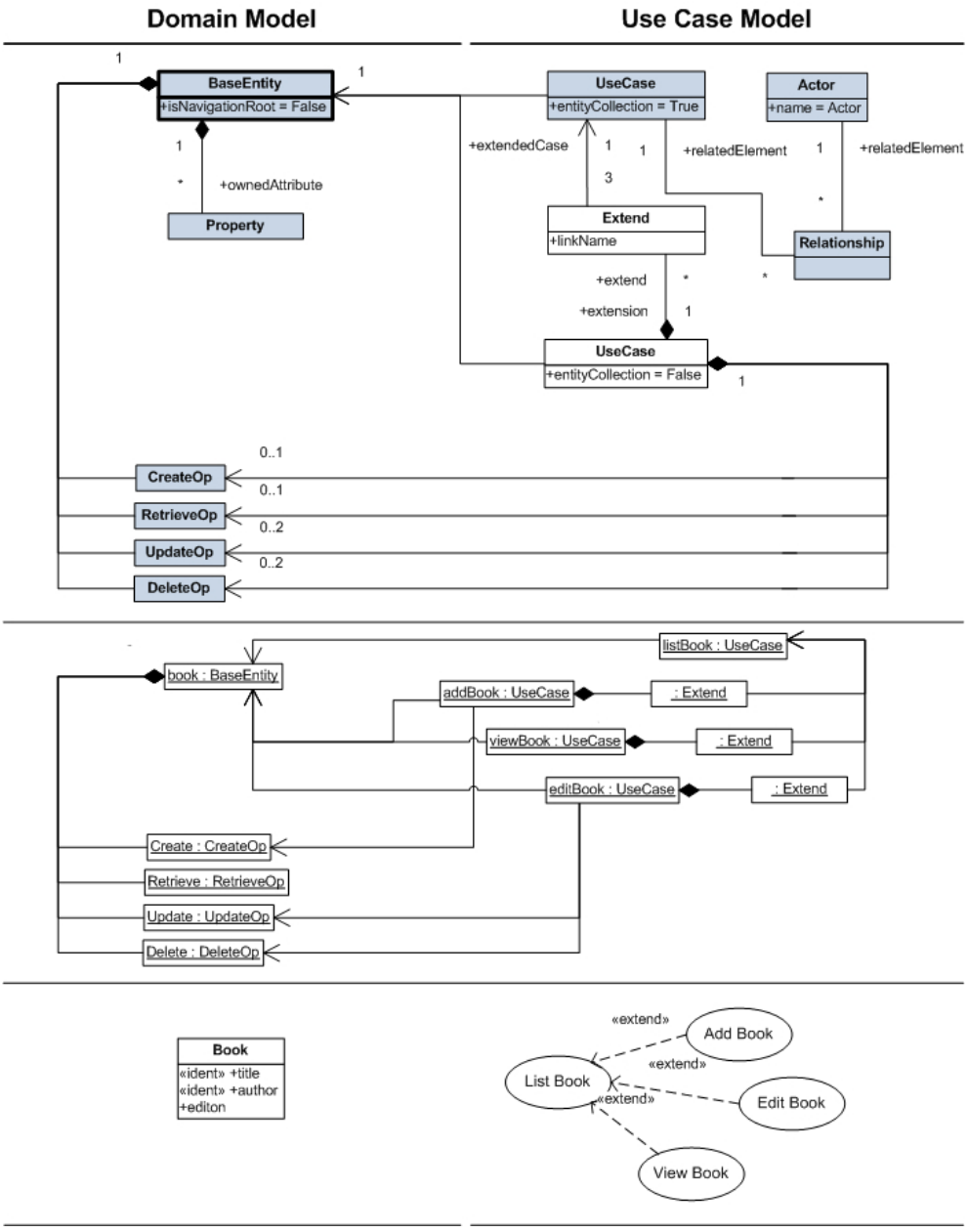

Figure 5.29: DM2UCM02: Transforming directly accessible base entities to UCM's CRUD UCs.

In Figs. 5.31 and 5.27, see, for example, use case "Edit Book", which includes use case "List Related bookCopies", which, in turn, is extended by use cases for adding and editing a BookCopy instance.

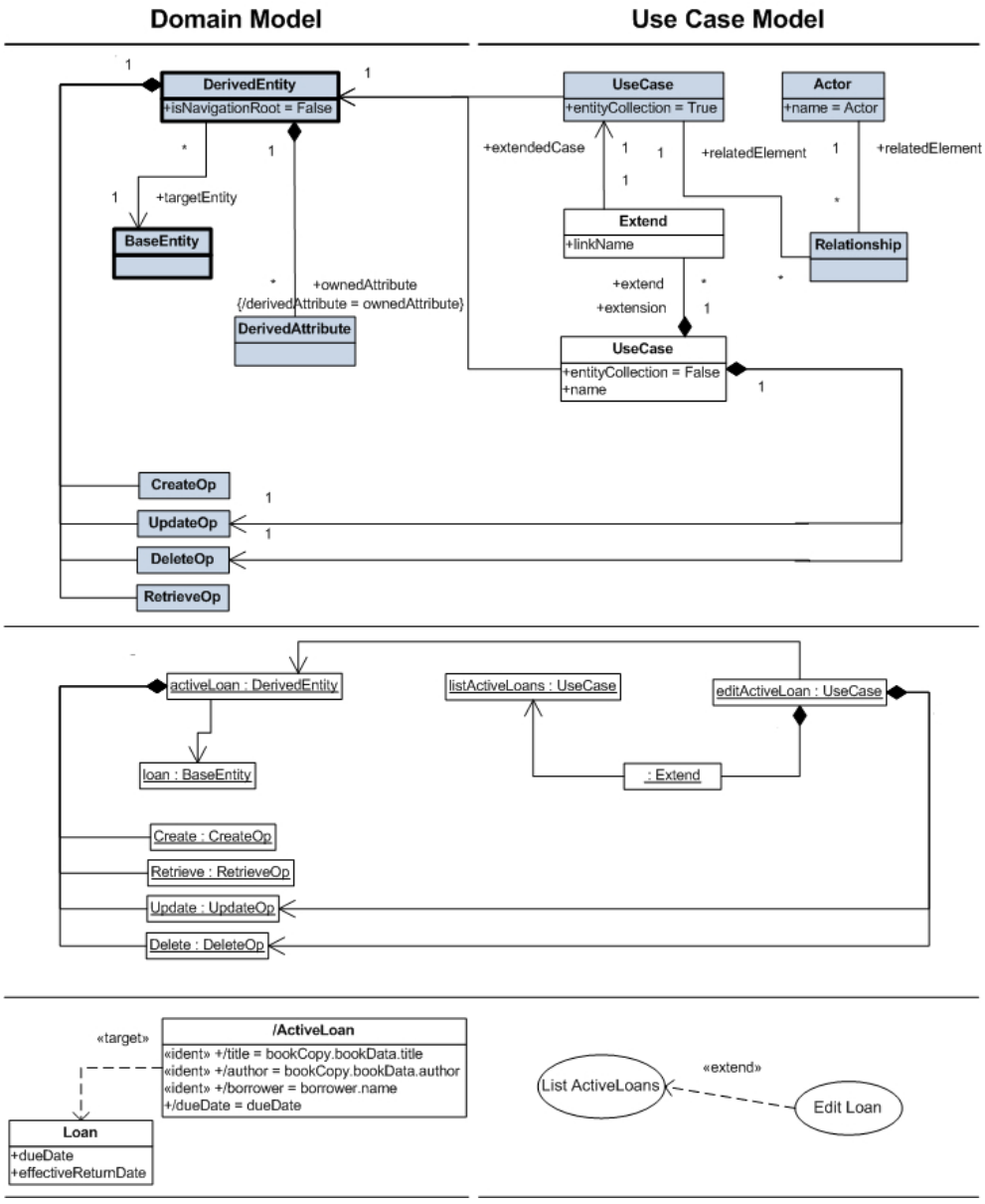

Figure 5.30: DM2UCM03: Transforming directly accessible derived entities to UCM's CRUD UCs.

5.4.5 DM2UCM05: Transform "to-many" relations from independent instances to Select Related UCs

Fig. 5.32 shows the rule for generating CRUD related use cases from "to-many" relations having an independent instance, that is "one-to-many" relations where the entity at the one side is optional, or "many-to-many" relations.

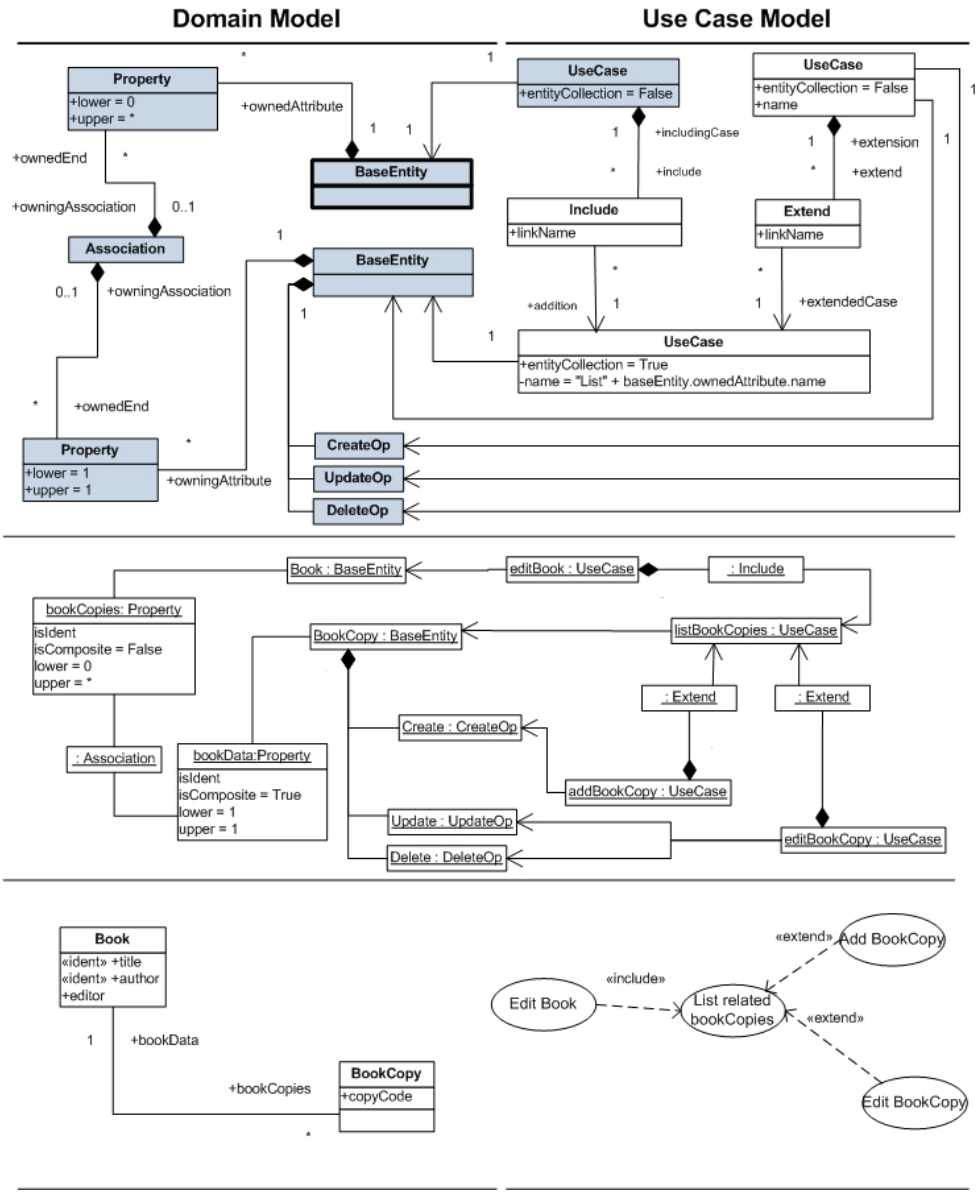

Figure 5.31: DM2UCM04: Transforming "to-many" relations from dependent instances.

5.4.6 DM2UCM06: Transform "to-one" relations from dependent or independent instances to Select Related UCs

A CRUD Entity use case shall include use cases that allow the selection of one related entity, when the UC's entity is "to-one" related to the same or another entity (see rule in Fig. 5.33).

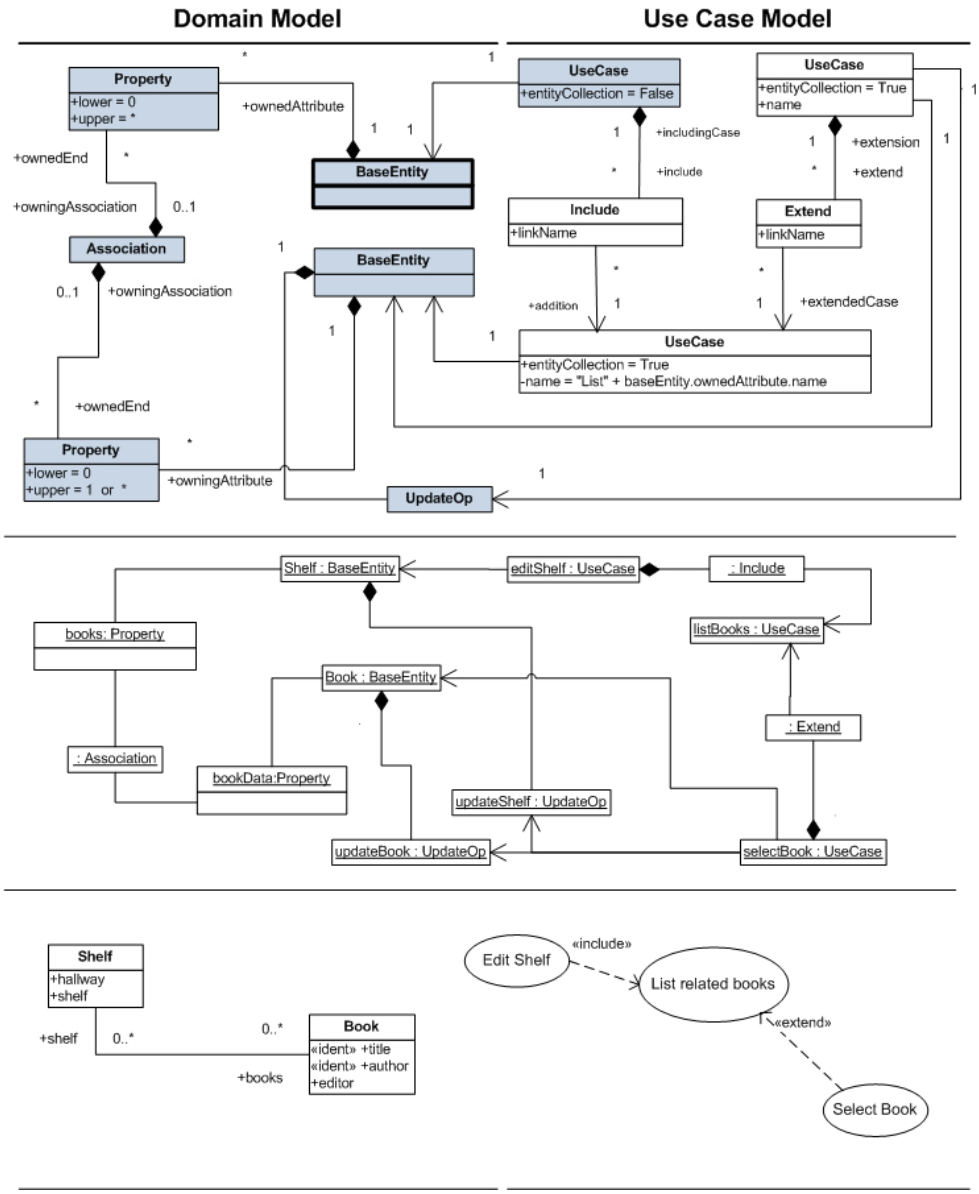

Figure 5.32: DM2UCM05: Transforming "to-many" relations from independent instances.

In our example, Fig. 5.27, this is the case of use cases "Select Borrower" and "Select BookCopy", which extend use cases "Edit Loan" and "Add Loan".

5.4.7 DM2UCM07: Transform user defined operations

Fig. 5.34 depicts the rule for generating a use case associated to a user defined operation. This kind of use case is included in CRUD use cases, or CRUD related

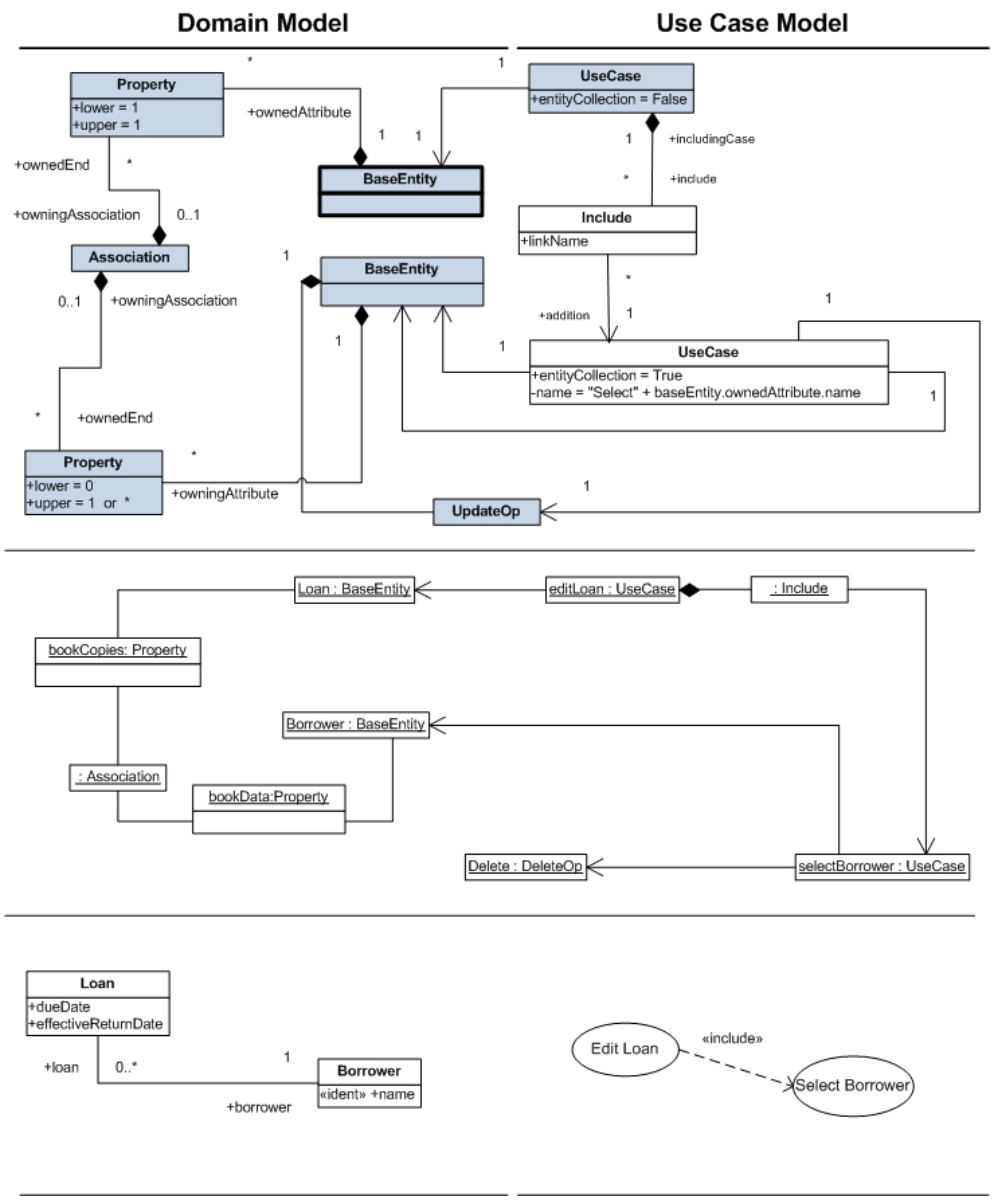

Figure 5.33: DM2UCM06: Transforming "to-one" relations from dependent or independent instances to Select Related UCs.

use cases, associated to the entity containing the operation definition.

5.5 Conclusions

This chapter addressed the model-to-model transformation rules for deriving a default UIM from the DM alone, DM2UIM transformation process (section 5.2),

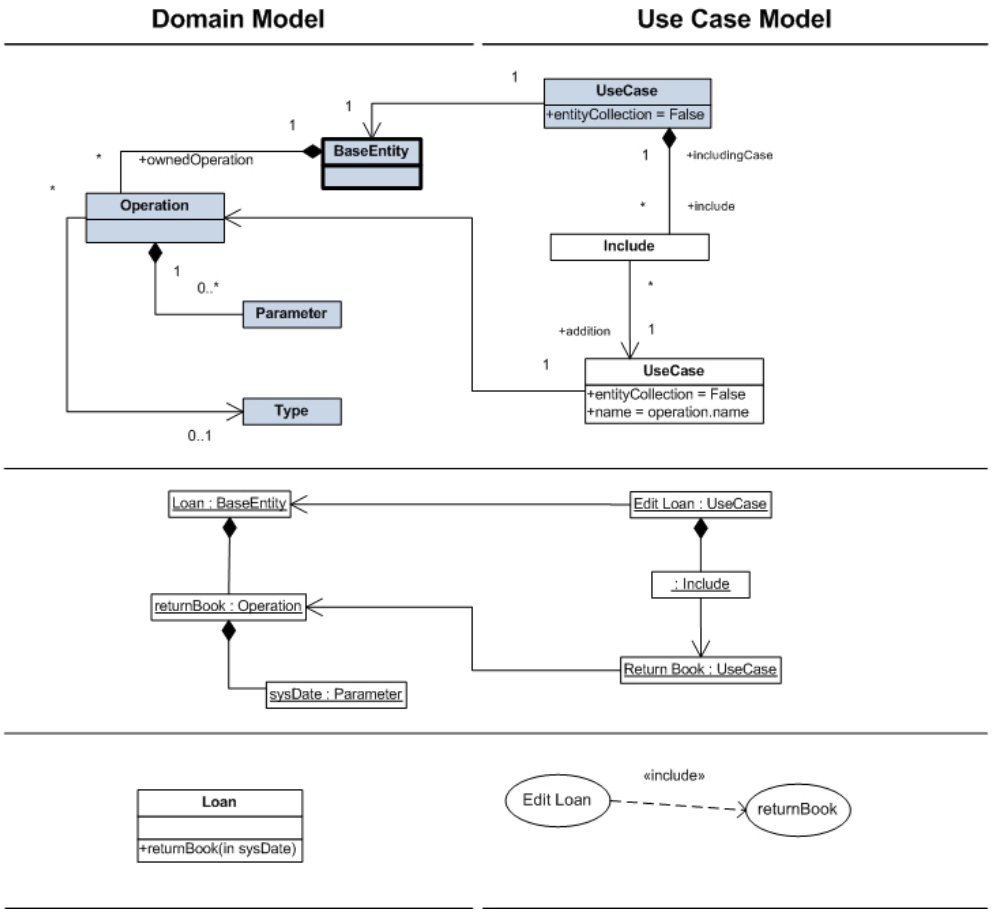

Figure 5.34: DM2UCM07: Transforming user defined operations to UCM.

and from the integrated UCM and DM, DM+UCM2UIM transformation process (section 5.3). Seven declarative transformation rules have been defined for the DM2UIM process, and eleven declarative transformation rules have been defined for the DM+UCM2UIM transformation process.

Also the model-to-model transformation rules for deriving an initial UCM from a DM, DM2UCM transformation process, have been addressed (section 5.4). Seven declarative transformation rules were defined for process DM2UCM.

The UIM derivation transformation rules are implemented imperatively in a proof-of-concept tool developed in C#, and presented in chapter 6.

Chapter 6

Implementation and Validation

This chapter presents the implementation of a proof-of-concept tool, two case studies, and discusses results obtained. Before ending, the chapter assesses the satisfaction of the proposed research goals.

6.1 Introduction

In order to be able to validate the approach, proposed in chapters 4 and 5, to the model-driven development of data-intensive interactive applications, the following sections address the implementation of a proof-of-concept tool and present the results of two case studies.

6.2 Proof-of-concept tool implementation

The proof-of-concept tool has been developed in C♯, and works with abstract model representations. It takes a domain model (DM) and optionally a use case model (UCM), asks the user for the model-to-model transformation process that shall be applied, DM2UIM or DM+UCM2UIM, applies the appropriate transformation rules, thus obtaining a UIM, and then applies a model-to-code transformation process that takes the DM and the UIM and generates XUL UI descriptions, Javascript for verifying constraints and executing operations and triggers, and RDF for persisting instances. The generated interactive application prototype runs with xulrunner [1] or Firefox [2].

The tool works with the metamodels abstract syntax, not having to transform the DM and UCM concrete syntax, that can be mostly obtained through any ordinary UML modeling tool, to the respective abstract notation. This way, we don't deviate from the focus of this dissertation, and the conformance of the models to the metamodels is guaranteed, as it is needed to explicitly create instances of the

[1] https://developer.mozilla.org/en/XULRunner
[2] https://developer.mozilla.org/en/XUL

metamodel's classes in order to produce a model abstract representation. All the metamodel concepts and concept relationships are present in the implemented tool.

6.2.1 Tool architecture

The prototyped tool is structured in the following components, which are also illustrated in Fig. 6.1:

- DM Metamodel
- UCM Metamodel
- UIM Metamodel
- DataComponent
- M2Mtransformations
- codeGen (M2C)
- MainApp

The first three components have the definition of the metamodels for the DM, the UCM and the UIM, respectively.

The DataComponent library has the definition of a metamodel for data resources, which enables the ulterior code generation for data persistence with RDF (Resource Description Framework).

The M2Mtransformations component is responsible for the model-to-model transformations that lead to the automatic generation of a UIM. It has two main methods that provide DM2UIM and DM+UCM2UIM transformation processes (see Fig. 6.1).

The codeGen component is in charge of the model-to-code transformations from the available models to the final code (see below "Architecture of the generated prototype").

Finally, the MainApp is simply a console application for orchestrating the process flow.

Domain model specification

The specification of the domain model to feed the prototyped tool is made by directly constructing the objects structure, as can be seen in the excerpts from the LibrarySystem case study (refer to section 6.4), that are presented below, together with the domain model for the enhanced LibrarySystem that is presented in Fig. 6.3.

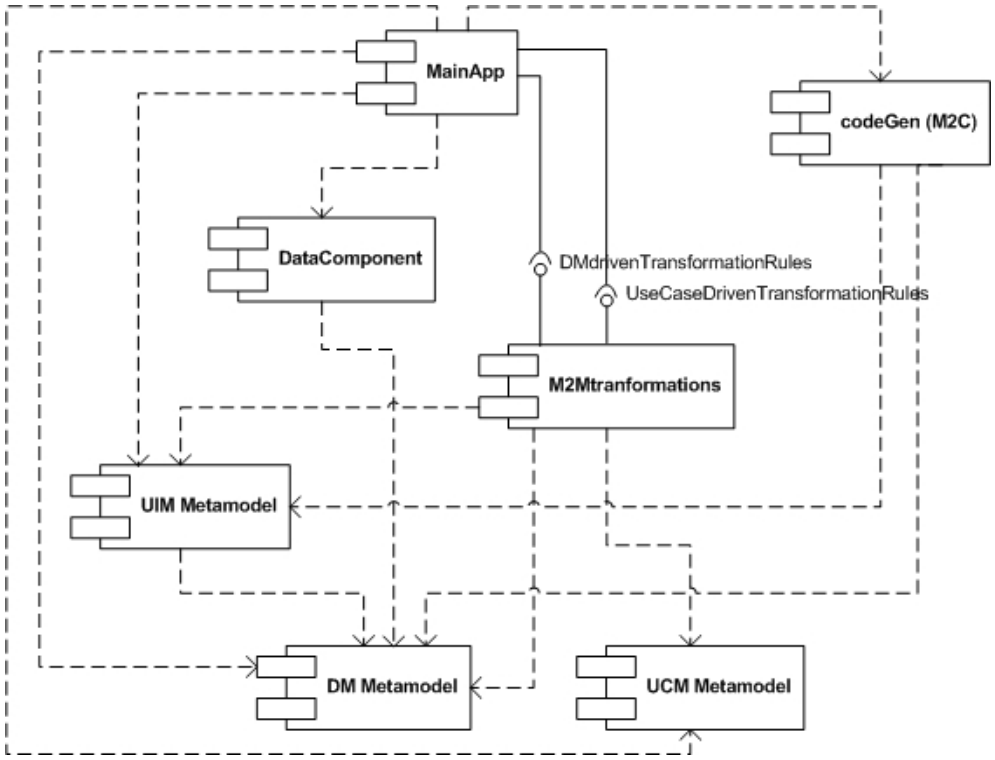

Figure 6.1: Proof-of-concept prototyped tool's components.

The way to create a new domain model is to directly instantiate a *umlDomainModel* class:

```
DomainModel umlDomainModel = new DomainModel();
```

And then add all desired entities, as shown below for base entities *Book* and *BookCopy*:

```
umlDomainModel.UMLDomainClasses.Add(new BaseEntity("Book",
     new Property(Enum_AttributeStereotype.Ident,
                  Enum_AccessModifier.public_access,
                  Enum_UMLType.String, "isbn"),
     new Property(Enum_AttributeStereotype.Ident,
                  Enum_AccessModifier.public_access,
                  Enum_UMLType.String, "title"),
     new Property(Enum_AttributeStereotype.Ident,
                  Enum_AccessModifier.public_access,
                  Enum_UMLType.String, "author"),
     new Property(Enum_AccessModifier.public_access,
                  Enum_UMLType.Integer, "edition"),
     new Property(Enum_AccessModifier.public_access,
                  Enum_UMLType.Integer, "year")));
```

Note that *BookCopy* has a derived attribute that references the corresponding

book title.

```
List<string> pathToTitle = new List<string>();
pathToTitle.Add("bookData");
pathToTitle.Add("title");
//
umlDomainModel.UMLDomainClasses.Add(new BaseEntity("BookCopy",
     new Property(Enum_AttributeStereotype.Ident,
                  Enum_AccessModifier.public_access,
                  Enum_UMLType.String, "copyCode"),
     new DerivedAttribute(Enum_AttributeStereotype.Ident,
                  Enum_AccessModifier.public_access,
                  Enum_UMLType.String, "bookTitle", pathToTitle)));
```

Associations between base entities are accomplished as shown below for the composition association between classes *Book* and *BookCopy*:

```
umlDomainModel.UMLClassRelationships.Add(
     new UMLRelationship(umlDomainModel.Find("Book"),
                         umlDomainModel.Find("BookCopy"),
                         "bookData",
                         "bookCopies",
                         Enum_Multiplicity.one_to_many,
                         Enum_TypeOfRelation.composition));
```

Base entity *Loan*:

```
umlDomainModel.UMLDomainClasses.Add(new BaseEntity("Loan",
     new Property(Enum_AccessModifier.public_access,
                  Enum_UMLType.String, "dueDate"),
     new Property(Enum_AccessModifier.public_access,
                  Enum_UMLType.String, "effectiveReturnDate")));
```

Has a user defined operation $returnBook(\ldots)$, that has one parameter, $sysDate$:

```
List<Parameter> parametros = new List<Parameter>();
parametros.Add(new Parameter("sysDate", Enum_UMLType.String));
//
SequenceOfActions bodyActions = new SequenceOfActions();
bodyActions.actionsBlock.Add(new Assign(new StateExp("effectiveReturnDate"),
     new ParameterExp("sysDate", typeof(Int32))));
bodyActions.actionsBlock.Add(new Assign(
     new AssociationEndCallExp("Loan", "status"),
     new AttributeCallExp(umlDomainModel.Find("LoanStatus"), "Inactive")));
ApplyFunctionAction invoKeLoaNxPersist = new ApplyFunctionAction("xpersist();");
invoKeLoaNxPersist.context = umlDomainModel.Find("Loan");
bodyActions.actionsBlock.Add(invoKeLoaNxPersist);
//
Operation methReturnBook = new Operation(Enum_AccessModifier.public_access,
     Enum_UMLType.Integer,
     "returnBook",
     parametros,
     bodyActions);
//
umlDomainModel.Find("Loan").addMethod(methReturnBook);
```

This corresponds to the following concrete syntax:

Context Loan::returnBook(in sysDate: Date)
body: self.update(
effectiveReturnDate = sysDate,
status = LoanStatus.Inactive
)

Operation *returnBook* has a precondition:

```
methReturnBook.pre_condition =
     new LogicOpExp(
          new RelationalOpExp(
               new AssociationEndCallExp("Loan", "Borrower"),
               RelationalOp.NEQ,
               new Value(Enum_UMLType.String, "")),
          LogicOp.AND,
          new RelationalOpExp(
               new AssociationEndCallExp("Loan", "BookCopy"),
               RelationalOp.NEQ,
               new Value(Enum_UMLType.String, "")));
```

Entity *BookCopy* has an invariant constraint stating that all bookcopy codes are unique:

```
umlDomainModel.Find("BookCopy").addConstraint(
    new UMLClassConstraint(
        "Copycodes are unique!",
        umlDomainModel.Find("BookCopy"),
        OCLQuantifiedExpression.forAll("x", "y",
            new AllInstancesCallExp("BookCopy"),
            new LogicOpExp(
                new RelationalOpExp(
                    new VariableExp("x",umlDomainModel.Find("BookCopy")),
                    RelationalOp.NEQ,
                    new VariableExp("y", umlDomainModel.Find("BookCopy"))),
                LogicOp.IMPLIES,
                new RelationalOpExp(
                    new VarAttributeCallExp("x", "BookCopy", "copyCode"),
                    RelationalOp.NEQ,
                    new VarAttributeCallExp("y", "BookCopy", "copyCode"))))));
```

And the corresponding concrete syntax:

Context BookCopy inv Copycodes_are_unique:
BookCopy −>forall(x, y | x ≠ y implies x.CopyCode ≠ y.CopyCode)

Use case model specification

Similarly, the specification of the use case model is also achieved by directly constructing the objects structure, as can be seen in the following excerpt from the LibrarySystem example.

Creating a new use case model:

```
UseCaseModel useCaseModel = new UseCaseModel();
```

Creating the actors:

```
Actor actor1 = new Actor("Librarian");
Actor actor2 = new Actor("Borrower");
useCaseModel.actors.Add(actor1);
useCaseModel.actors.Add(actor2);
```

Creating the packages that will contain use cases:

```
UseCasePackage ucp1 = new UseCasePackage("Manage Books");
UseCasePackage ucp2 = new UseCasePackage("Manage Loans");
```

Creating use case *Add Loan* associated to the base entity *Loan*:

```
UseCase ucAddLoan = new UseCase("Add Loan");
ucAddLoan.associatedClass = umlDomainModel.Find("Loan");
ucAddLoan.associatedCRUDOpers.Add(Enum_CRUD.Create);
ucp2.addUseCase(ucAddLoan);
```

Linking an actor to a use case:

```
useCaseModel.linksActorUseCase.Add(new LinkActorUseCase(actor1, ucAddLoan));
```

Creating use case *Edit Book* associated to the base entity *Book*:

```
UseCase ucEditBook = new UseCase("Edit Book");
ucEditBook.associatedClass = umlDomainModel.Find("Book");
ucEditBook.associatedCRUDOpers.Add(Enum_CRUD.Update);
ucp1.addUseCase(ucEditBook);
```

And finally, creating use case *List BookCopies*, and including it in use case *Edit Book*:

```
UseCase ucListBookCopies = new UseCase("List BookCopies");
ucListBookCopies.associatedClass = umlDomainModel.Find("BookCopy");
ucp1.addUseCase(ucListBookCopies);
//
UseCaseInclude uci2 = new UseCaseInclude(ucEditBook, ucListBookCopies);
```

6.2.2 Architecture of the generated prototype

The output of the sequentially applied M2M and M2C processes, in the prototyped tool, comprises:

- XUL files, which contain the description of the UI elements without any look & feel detail;

- Javascript files that contain the CRUD operations, user defined operations, and validation routines for invariant and precondition constraints and for

validating that characters inputted in a field match the type of the DataAIO that originated it;

- A RDF file, that contains information about entity instances and its relationships to other instances.

```
<RDF:Description
   RDF:about="http://www.class_Book.com/
Book_1229284976700"
   Book:ISBN="18752-726"
   Book:Title="Equador"
   Book:Author="Miguel S. Tavares"
   Book:Edition="9"
   Book:Year="2004" />
```

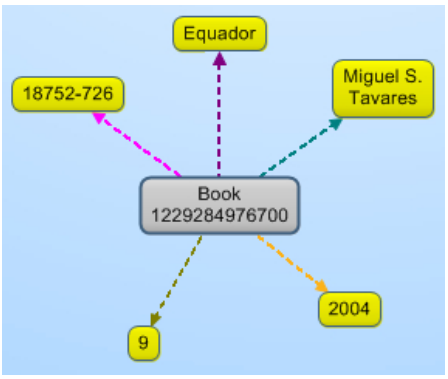

Figure 6.2: Example of an RDF description and the corresponding conceptual graph displayed with Gruff (http://www.franz.com/agraph/gruff).

Fig. 6.2 shows an example of an RDF description of an object and the corresponding graph notation that it conceptually represents. For easing the manipulation of RDF datasources and resources, we resorted to a library, rdfds.js[3], that provides javascript objects. Namely, it contains four objects, RDFDataSource, RDFNode, RDFLiteral, and RDFEnumerator. An RDF DataSource is a graph of nodes and literals. The constructor for RDFDataSource takes one argument, a URI of an RDF file to use. If the URI exists, the contents of the RDF file are loaded. If it does not exist, resources can be added to it and then written using a save method. If the URL argument is null, a blank datasource is created.

6.3 Model-to-code mappings from UIM to XUL

As aforementioned, the prototyped tool sequentially applies a M2M transformation process (DM2UIM or DM+UCM2UIM) and then a M2C transformation that, finally, yields an interactive executable prototype comprising a set of XUL and Javascript files and an RDF file. This way, regarding the user interface, a set of model-to-code transformation rules are needed for generating XUL from the available models. Table 6.1 illustrates the mappings between UIM elements and XUL code excerpts.

The generated XUL descriptions, and Javascript files, are interpreted by xulrunner in order to render the UI and execute the prototype.

[3]http://cvs.zope.org/Packages/Moztop/moztop/content/Inspector/Attic/rdfds.js

UIM concrete patterns	XUL UI description

InteractionSpace	`<?xml version="1.0"?>`
	`<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>`
	`<window`
	`id="..."`
	`title="..."`
	`width="400" height="500"`
	`onload="init();"`
	`orient="horizontal,sizeToContents"`
	`scroll="true"`
	`xmlns="http://www.mozilla.org/keymaster/gatekeeper/`
	`there.is.only.xul">`
	`...`
	`</window>`

InteractionBlock	`<groupbox>`
	`<caption label="..."/>`
	`...`
	`</groupbox>`

ViewList ViewRelatedList	`<groupbox id="..." collapsed="true">`
	`<caption label="..." />`
	`<listbox id="list..." rows="10" seltype="single"`
	`onselect="selectItem...(selectedIndex);" >`
	`...`
	`</listbox>`

MenuBar, Menu and MenuItems	`<menubar id="main-menubar" flex="1">`
	`<toolbarseparator />`
	`<menu label="..." accesskey="..." >`
	`<menupopup>`
	`<menuitem name="..." label="..."`
	`oncommand="opcao_...();" />`
	`<menuitem name="..." label="..."`
	`oncommand="opcao_...();" />`
	`<menuitem name="..." label="..."`
	`oncommand="opcao_...();" />`
	`</menupopup>`
	`</menu>`
	`</menubar>`

DataAIO	`<row>`
	`<label control="..." value="..." />`
	`<textbox id="..." value=""/>`
	`</row>`

DataAIO when inside ViewList or ViewRelatedList	`<listitem>`
	`<listcell label="..."/>`
	`<listcell label="..."/>`
	`<listcell label="..."/>`
	`</listitem>`

ActionAIO	`<button id="..." label="..." oncommand="call_...();" />`

Table 6.1: Illustration of M2C transformations between the UIM and XUL.

6.4 Case study 1 - Library System

This section analyzes the results of the LibrarySystem case study obtained by the aforementioned prototyped tool for the steps it is able to automate.

6.4.1 Library System Domain Model

Figure 6.3 shows a LibrarySystem domain model a little different from the one showed in previous chapters' examples.

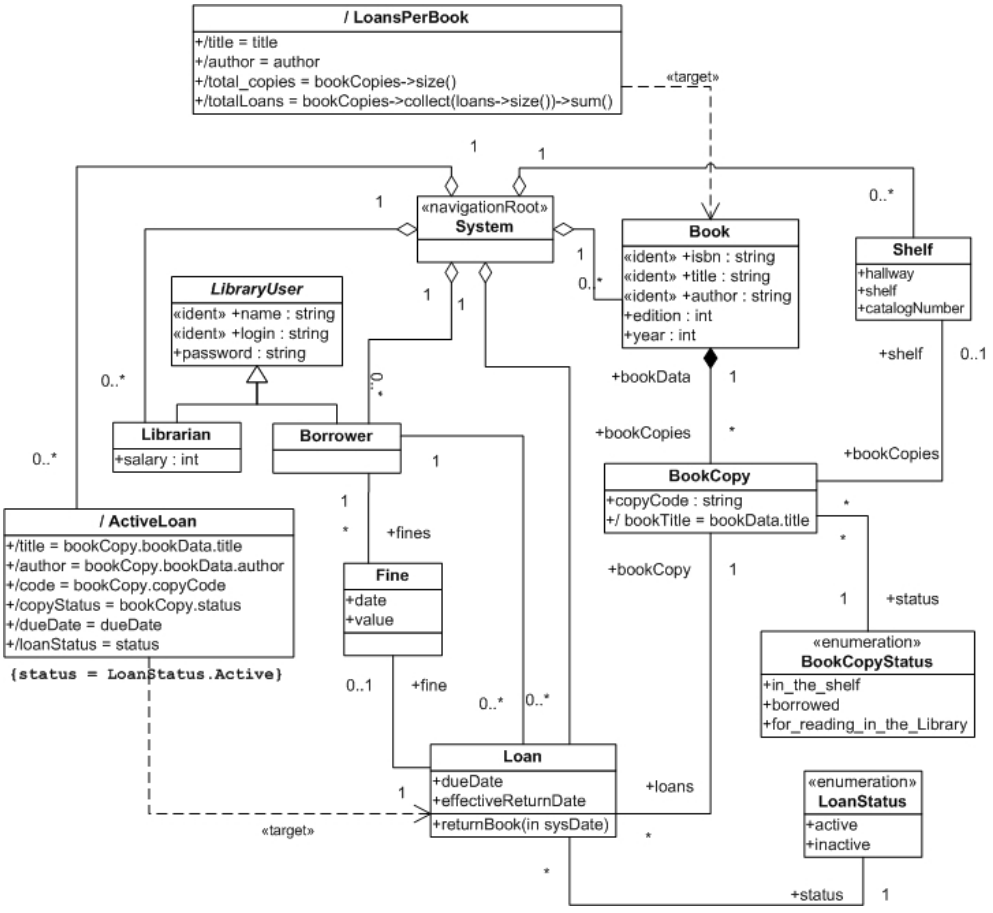

Figure 6.3: Domain model for a Library Management System (LibrarySystem).

The illustrated domain model shows us a Library System comprising books (*Book*), each collecting information (isbn, title, author, edition and year) of a set of book copies (*BookCopy*). A book copy may be stored in a shelf (*Shelf*), which, in turn, may have several book copies. A book copy has a status of enumerable type *BookCopyStatus*. A book copy may be involved in loans (*Loan*).

Librarians (*Librarian*) and borrowers (*Borrower*) are library users (*LibraryUser*). Borrowers may borrow books (*Loan*). A loan involves one borrower and one book copy, and may be in an active or inactive status (*LoanStatus*).

When a lent book copy is delivered after the due date, a EUR 1 fine (*Fine*) must be automatically created, and this is specified through the following trigger:

Context Loan trigger after update:
effect: if (self.effectiveReturnDate < self.dueDate)
Fine.create(
date = self.effectiveReturnDate,
value = 1,
borrower = self.borrower,
loan = self
)

Operation *returnBook* in class *Loan* is defined as:

Context Loan::returnBook(in sysDate: Date)
body: self.update(
effectiveReturnDate = sysDate,
status = LoanStatus.Inactive
)

The following invariants are also defined:

Context Book inv : self.title ≠ “”

Context Book inv : self.author ≠ “”

Context BookCopy inv Copycodes_are_unique:
BookCopy.allInstances() −>forall(x, y | x ≠ y implies x.CopyCode ≠ y.CopyCode)

Context LibraryUser inv : self.login ≠ self.name and self.login ≠ self.password

Derived entities, or business views, *LoansPerBook* and *ActiveLoan*, are also defined in the domain model. *ActiveLoan* is a derived entity that targets base entity *Loan* (refer to section 4.4.2), and has an invariant that filters Loan's instances. Basically, an active loan is a loan with *status* = *LoanStatus.Active*. All its attributes are derived and refer to Loan or Loan related entities' attributes.

LoansPerBook is a derived entity targeting base entity *Book*, with no filtering invariant. Its attributes either reference attributes of *Book* (*title* and *author*) or calculate a value by making use of an OCL expression. The latter is the case of attribute *total_copies*, that counts the number of *BookCopy* instances related to a *Book*, and *totalLoans*, which calculates the total loans related to the book copies of a given book.

6.4.2 Generated Prototype after process DM2UIM

The prototyped generator tool sequentially applies DM2UIM process and the model to code generation process for XUL, Javascript and RDF, to the LibrarySystem domain model.

The DM2UIM process derives a UIM based only in the domain model depicted in figure 6.3 and in the rules defined in section 5.2.

The applied code generation process generates files with UI XUL descriptions, Javascript code-behinds and RDF.

To analyze the result of applying the DM2UIM process on the LibrarySystem domain model, we will focus on a subset of its classes. By applying rule DM2UIM01 (refer to section 5.2) to class Book, an interaction space showing the books own attributes is created. Through rule DM2UIM05, Book's relation to class BookCopy becomes apparent, by generating a list of related book copies, from where the user is able to create new or edit existing related bookcopies. Figure 6.4 shows the canonical abstract prototype, which represents the UIM element, and the rendering of the XUL description file generated from the Book interaction space, which was, in turn, generated from the class Book and its relations.

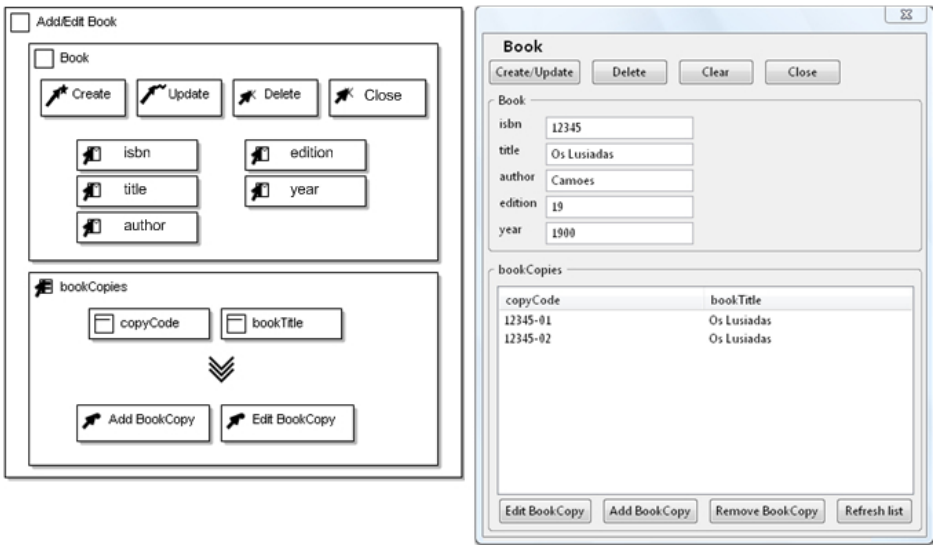

Figure 6.4: CAP and screenshot of executing prototype for Book interaction space.

Figure 6.5 shows the same results obtained from class BookCopy and its relations to other classes. Notice that class BookCopy has relations to Shelf and Loan, and that it has attribute *status* of enumerated type *BookCopyStatus*.

Figure 6.6 shows part of the screenshots attained from the execution of the generated prototype, which was obtained after applying DM2UIM and M2C pro-

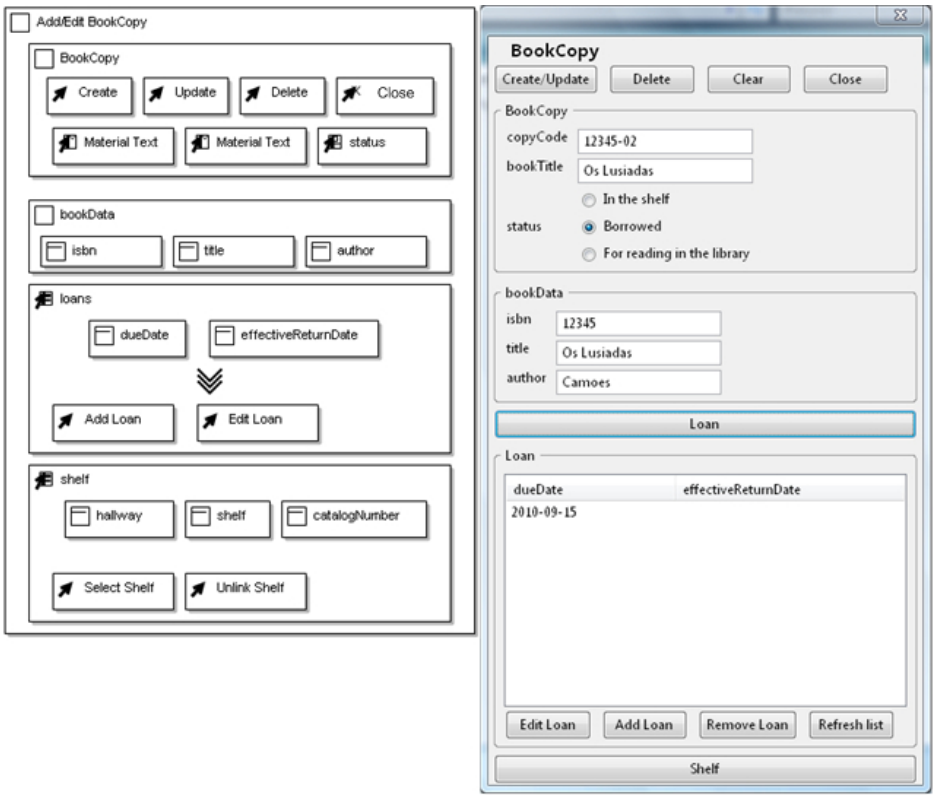

Figure 6.5: CAP and screenshot of executing prototype for BookCopy interaction space.

cesses, having been produced a set of XUL and Javascript files, and an RDF file for persisting the base entities' instances.

Note the difference between the UI elements generated from composition relations and from simple associations. An example of the former is the one-to-many relation between *Book* and *BookCopy* (see fig. 6.4), and of the latter is the one-to-many relation between *Shelf* and *BookCopy* (see fig. 6.6).

Notice also, in fig. 6.6, the screenshots for *Loan*, where it is possible to select a *BookCopy* and a *Borrower* instances. As mentioned in a previous chapter, this is due to the fact that the screenshots have been obtained when the user had navigated directly from the list of all loans (*LoanCollection*) to the edition of a given loan. If, for instance, the user had navigated to a loan from a related book copy instance, it would not be possible to select a different book copy. The same is true, if the user had navigated to the edition of a loan from a related instance of *Borrower*.

The windows generated from *ActiveLoan* and *LoansPerBook* derived entities can be seen in figure 6.7. As expressed before (see section 5.2.4), only the fields corresponding to attributes from the target base entity are updatable. All other

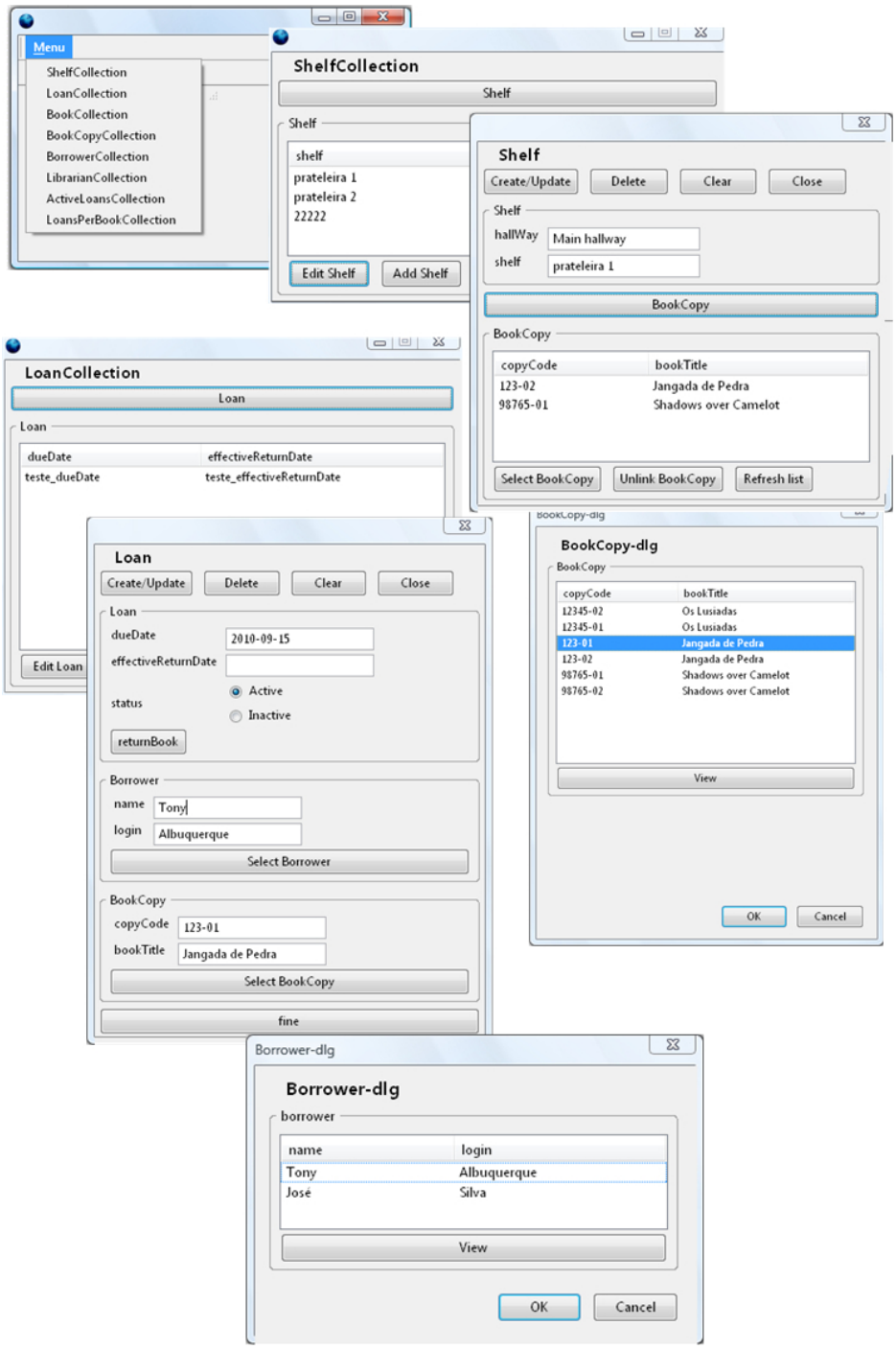

Figure 6.6: Partial set of screenshots of LibrarySystem executing prototype obtained after process DM2UIM.

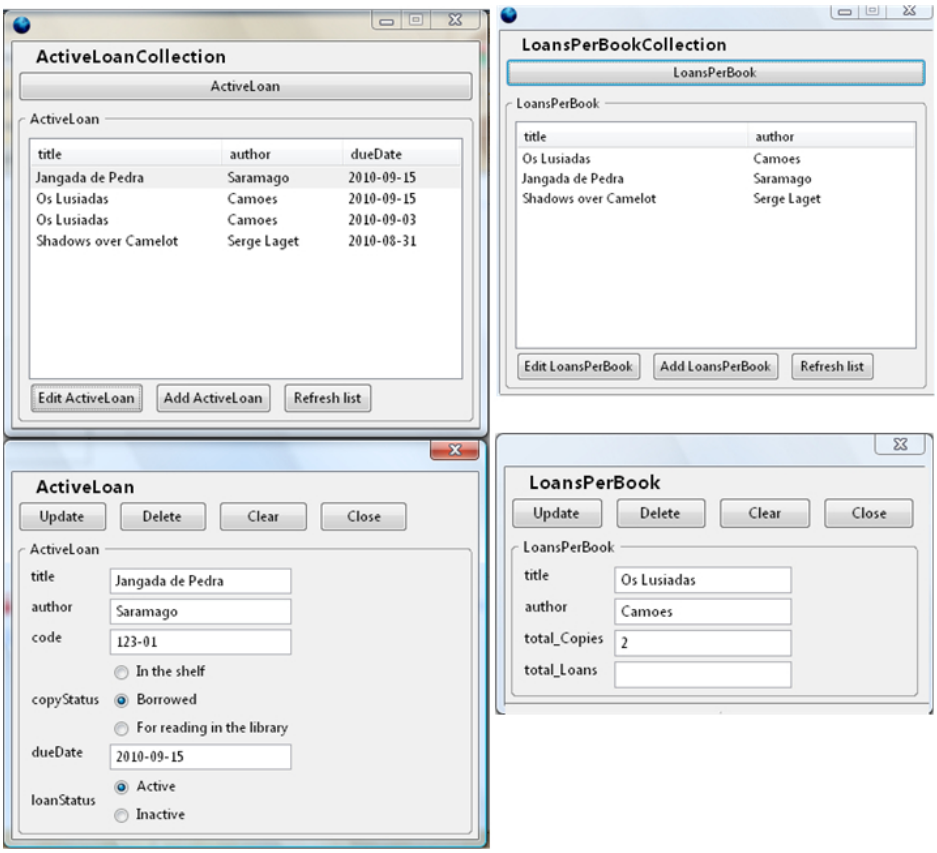

Figure 6.7: Screenshots of the windows generated from the two derived entities defined in the Library System domain model.

fields are read-only.

The prototyped code generation process generates Javascript code for verifying if data entered in fields are of the type defined for the corresponding attribute in the domain model. Figure 6.8 shows some of the windows that may appear to the user when datatype or invariants validation fails.

Below is an example of the Javascript code generated for validating data according to invariants defined over class *Book*:

```
function checkConstraint_1(msg)
{
  var ret = 1;
  var chxtitle = document.getElementById("title").value;
  if (!(chxtitle != ""))  {
      if (msg!=0)
        alert("Constraint violated: CONSTRAINT 1.");
      ret = 0;
  }
  return ret;
}
```

Figure 6.8: Screenshots of windows showing messages after validating data types or invariants.

```
function checkConstraint_2(msg)
{
  var ret = 1;
  var chxauthor = document.getElementById("author").value;
  if (!(chxauthor != ""))  {
      if (msg!=0)
        alert("Constraint violated: CONSTRAINT 2.");
      ret = 0;
  }
  return ret;
}
```

```
function checkConstraints(x)
{
//ret=1 --> OK;  ret=0 --> KO;
    var ret = 1;
    if (checkConstraint_1(x)==0) ret = 0;
    if (checkConstraint_2(x)==0) ret = 0;
    return ret;
}
```

```
function verifyConstraints()
{
  var createupdatebt = document.getElementById("createupdatebt");
  var deletebt = document.getElementById("deletebt");
  if (checkConstraints(0)==0) {
     createupdatebt.disabled = true;
     deletebt.disabled = true;
  }
  else{
     createupdatebt.disabled = false;
     deletebt.disabled = false;
  }
}
```

The invariants and fields' datatypes are only verified when the user executes modification operations, such as object creation of update, or the invocation of a user defined operation. If the constraints aren't verified then the buttons for initiating those operations are disabled.

Below is the code generated for invoking the operation returnBook, defined if class *Loan*, including the verification of the operation's pre-condition (it lacks the final invocation of the Fine's creation operation, due to incompleteness of the code generation prototype):

```
function handleClick_returnBook()
{
```

First, the operation's precondition is verified:

```
  if (checkPrecondition_returnBook(1) != 1) {
      alert("ALERT: Method's Precondition doesn't hold!!!");
      exit;
  }
```

Then, the provision of the input parameters is addressed, through an input parameters window:

```
  var inpdlgpar = { var_sysDate: null , ok: null };

  var inpdlg = window.openDialog(
          "chrome://sysapp/content/InputParametersSpace-returnBook.xul",
          "InputParametersSpace-returnBook",
          "chrome,modal,resizable,centerscreen", inpdlgpar);
```

Then, if the input parameters are valid, the operation is performed, and an output parameters window is opened, otherwise a cancellation message is provided to the user:

```
  var outdlgpar = { result: null, statusMessage: null };

  if (inpdlgpar.ok)
  {
     document.getElementById("effectiveReturnDate").value =
          inpdlgpar.var_sysDate;
     document.getElementById("status").selectedIndex = parseInt(1);
     xpersist();
     outdlgpar.statusMessage = "Operation sucessfully executed !";
     window.openDialog(
          "chrome://sysapp/content/OutputResultSpace-returnBook.xul",
          "OutputResultSpace-returnBook",
          "chrome,modal,resizable,centerscreen", outdlgpar);
  }
else{
     outdlgpar.statusMessage = "Operation cancelled !";
     outdlgpar.result = null;
     window.openDialog(
          "chrome://sysapp/content/OutputResultSpace-returnBook.xul",
          "OutputResultSpace-returnBook", "chrome,modal,resizable,centerscreen",
          outdlgpar);
    }
}
```

The function that verifies the operation's precondition:

```
function checkPrecondition_returnBook(msg)
{
  var ret = 1;
  var chxname = document.getElementById("name").value;
  var chxlogin = document.getElementById("login").value;
  if (!( (chxname != "") && (chxlogin != "") ))
  {
      if (msg!=0)
        alert("pre-condition failed - 1");
      ret = 0;
  }
  var chxcopyCode = document.getElementById("copyCode").value;
  var chxbookTitle = document.getElementById("bookTitle").value;
  if (!( (chxcopyCode != "") && (chxbookTitle != "") ))
  {
      if (msg!=0)
        alert("pre-condition failed - 2");
      ret = 0;
  }
  return ret;
}
```

And finally, the code generated for persisting (create or update) a loan, including the code from the update trigger, also defined in class *Loan*:

```
function xpersist()
{
```

First, the invariants are checked:

```
if (checkConstraints(1)!=1) {
    alert("There are constraints verified to be false!!!");
    exit;
}
```

Then, it is verified if it is happening a creation or an update, and the respective operation is performed (note that a "before" trigger is not defined):

```
var criarLoan = 1;
var node0 = ds.getNode("http://www.class_Loan.com/all-Loans");
var newNode0 = null;

if (vloan_RDF_ID != null)
{
  criarLoan = 0;
  newNode0 = ds.getNode(vloan_RDF_ID);
}
else
{
  criarLoan = 1;
  var vSeqelemDifDt = new Date();
  var elemDif = vSeqelemDifDt.getTime();
  vloan_RDF_ID = "http://www.class_Loan.com/Loan_" + elemDif;
  newNode0 = ds.getNode(vloan_RDF_ID);
  node0.addChild(newNode0);
  newNode0.addTargetOnce("http://www.class_Loan.com/rdf#RDF_ID",
                         "http://www.class_Loan.com/Loan_" + elemDif);
}

newNode0.addTargetOnce("http://www.class_Loan.com/rdf#dueDate",
                       document.getElementById("dueDate").value);

newNode0.addTargetOnce("http://www.class_Loan.com/rdf#effectiveReturnDate",
                       document.getElementById("effectiveReturnDate").value);

var selIndCreate = document.getElementById("status").selectedIndex;
newNode0.addTargetOnce("http://www.class_Loan.com/rdf#status",
                       selIndCreate.toString());

newNode0.addTargetOnce("http://www.class_Loan.com/rdf#Borrower",
                       vborrower_RDF_ID);
newNode0.addTargetOnce("http://www.class_Loan.com/rdf#BookCopy",
                       vbookcopy_RDF_ID);
```

Then, the code for "after" triggers is injected. In this case, it is defined an "after update" trigger :

```
  if (criarLoan == 1)
  {
  }
  else
  {
    var xfine = null;
    var xnode0 = ds.getNode("http://www.class_Fine.com/all-Fines");
    var xSeqelDfDt = new Date();
    var xelDft = xSeqelDfDt.getTime();
    var vFine_RDF_ID = "http://www.class_Fine.com/Fine_" + xelDft;

    var xfine = ds.getNode(vFine_RDF_ID);
    xnode0.addChild(xfine);
    xfine.addTargetOnce("http://www.class_Fine.com/rdf#RDF_ID", vFine_RDF_ID);

    var xLhsdateFine = document.getElementById("effectiveReturnDate").value;
    xfine.addTargetOnce("http://www.class_Fine.com/rdf#dateFine", xLhsdateFine);

    var xLhsvalueFine = "1";
    xfine.addTargetOnce("http://www.class_Fine.com/rdf#valueFine",
                          xLhsvalueFine);

    xSelf_Loan = ds.getNode(vLoan_RDF_ID);

    var xLhsborrower = (xSelf_Loan.getTarget(
                          "http://www.class_Loan.com/rdf#Borrower")).getValue();
    xfine.addTargetOnce("http://www.class_Fine.com/rdf#Borrower", xLhsborrower);

    var xLhsloan = vLoan_RDF_ID;
    xfine.addTargetOnce("http://www.class_Fine.com/rdf#Loan", xLhsloan);

//xCallOperationAction

  }
```

Finally, the entire data structure is saved in the RDF file:

```
ds.save();
}
```

6.4.3 Library System Use Case Model

Figure 6.9 and Table 6.2 respectively depict the use case model for the Library System, and the values of the meta-attributes that provide the integration between the DM and the UCM.

The use case model and the respective meta-attributes values, which may be defined through tagged-values, show that the modeled library system has two actors (*Librarian* and *Borrower*). Use cases are packaged into four packages, three containing use cases accessible only to actor *Librarian* and one has use cases for *Borrower*.

Librarian directly accesses use cases “List Books”, “List Loans”, “Add Loan” and “List Borrowers”, from where he can add or edit new or existing appropriate instances.

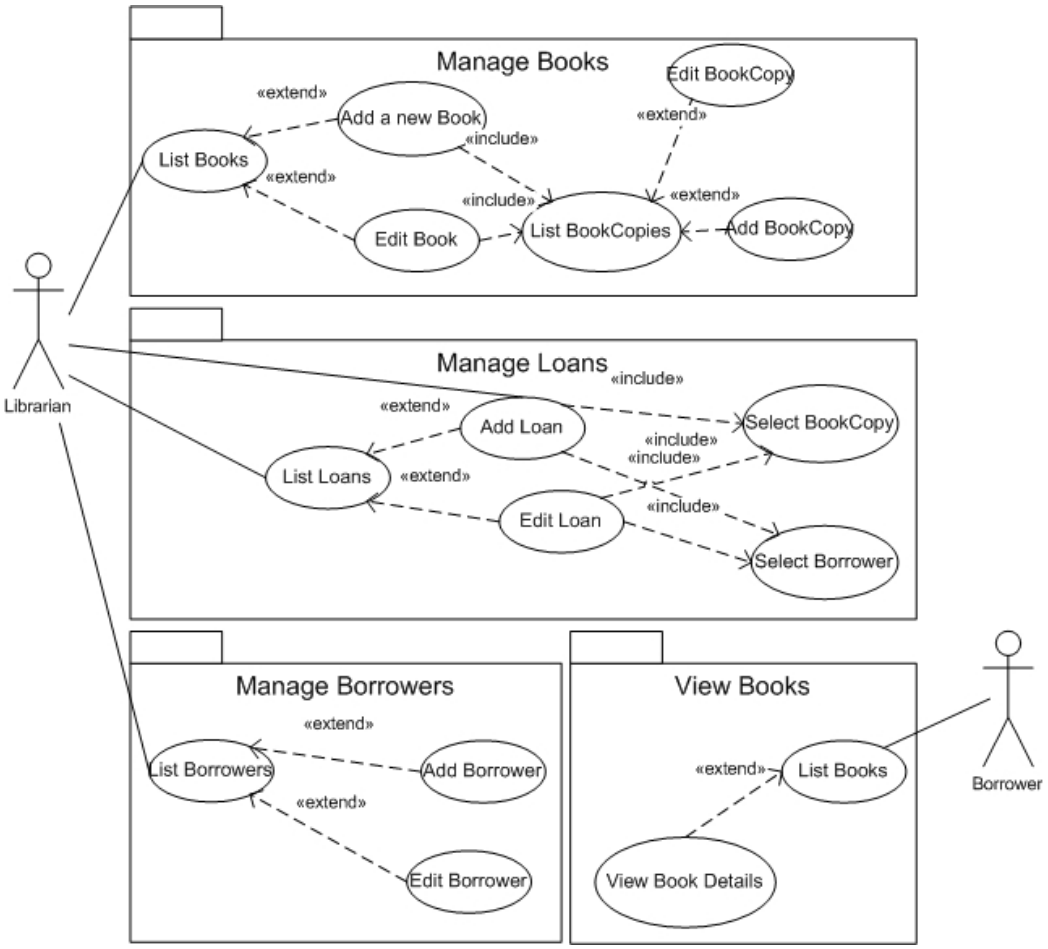

Figure 6.9: Partial use case model (UCM) for the Library Management System.

Borrower is only able to directly access "List Books" from where he can only view the details of the selected book.

6.4.4 Generated Prototype after process DM+UCM2UIM

When applying process DM+UCM2UIM, followed by the code generation process, to the previously presented Library system domain and use case models, the result is a prototype that maps the functionality defined in the use case model. As before, the interaction spaces' structure is mapped from the domain model.

This is apparent in Figures 6.10 and 6.11, which show a part of the screenshots attained from the execution of the generated prototype, obtained after applying DM+UCM2UIM and M2C processes.

Figure 6.10 shows the screenshots of the initial window and of each actor's main window, showing the generated menus and menu options. These are ob-

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

Table 6.2: Entities and operations associated (via tagged values) with the use cases in Fig. 6.9.

tained after application of rule DM+UCM2UIM01. Note that, contrary to what we saw in section 6.4.2, where all the functionality of the system was available to the user, by using a UCM and process DM+UCM2UIM, the UI only makes available to the user the operations that were specified for the corresponding actor, in the use case model.

Figure 6.11 shows a partial result of applying the DM+UCM2UIM process on the LibrarySystem domain and use case models. The figure puts side by side the screens corresponding to a "Librarian" use cases "List Books" and "Edit Book" (on the left) and a "Borrower" use cases "List_Books" and "View Book Details" (on the right).

6.5 Case study 2 - Conference Review System

This section shows the results of the Conference Review System case study. As before, we will first analyze the results of applying process DM2UIM and after that the ones of applying DM+UCM2UIM. This case study was obtained by partially reverse engineering the MyReview conference management system [4].

This case study allows us to see how many-to-many relationships effect on the generated UI.

[4]http://myreview.sourceforge.net/

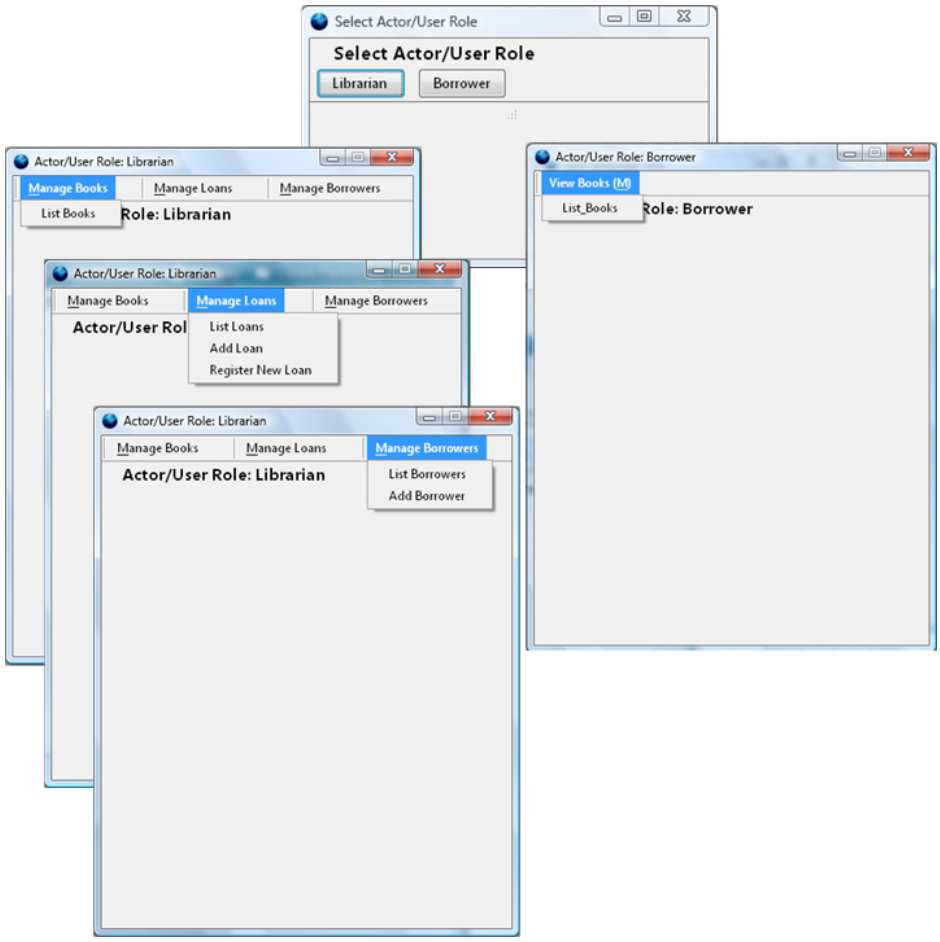

Figure 6.10: Screenshots of the initial window and of each actor's main window, showing the generated menus and menu options.

6.5.1 Conference Review System Domain Model

Figure 6.12 shows a simplified Conference Review System domain model.

The simplified conference review system, modeled in figure 6.12, allows for the submission of paper abstracts (*Paper_Abstract*), the establishment of reviewing assignments (*Assignment*), and the definition of reviews (*Review*), ratings (*Rating*) and review marks (*Review_Mark*) according to defined criteria (*Criterium*).

A paper abstract may have several authors (*Author*) and, in turn, each author may be related to several papers. A paper abstract may be classified in several topics (*Topic*), and each topic may be used by several papers.

A reviewing assignment is related to only one reviewer (*Reviewer*), but each reviewer may have several assignments.

Each paper abstract is related to several reviews, ratings and review marks.

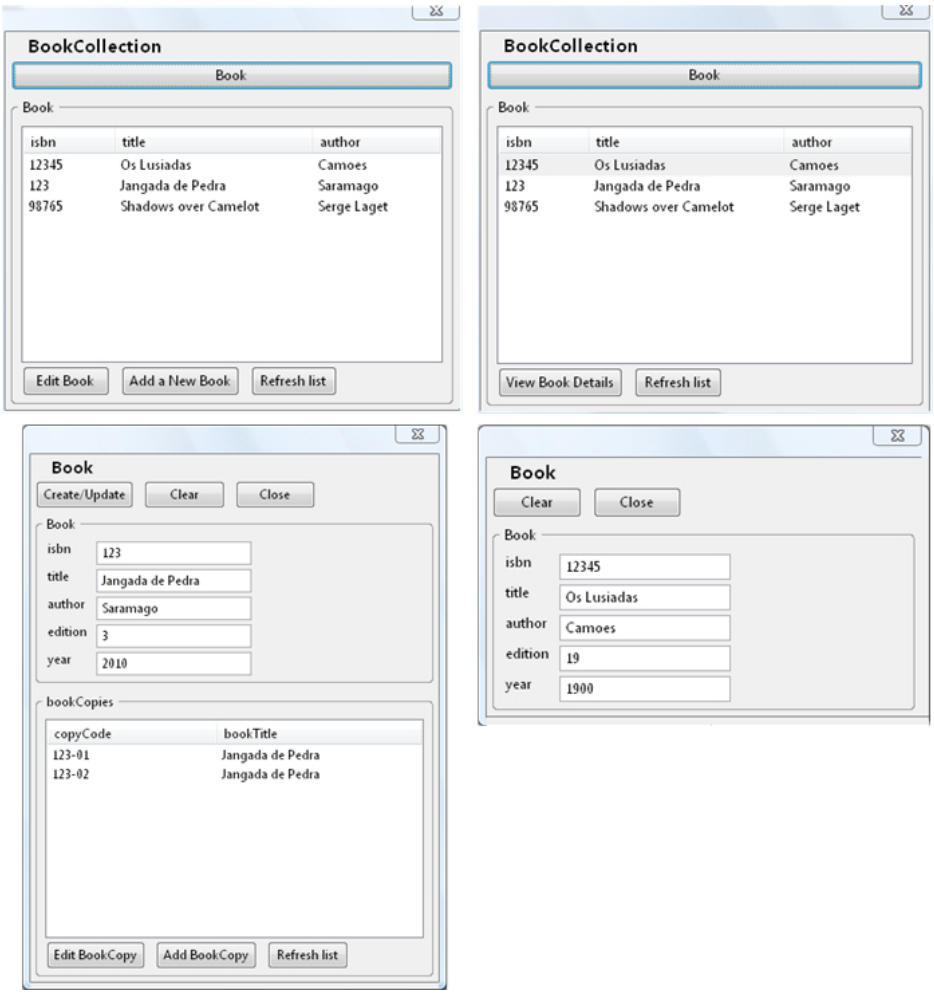

Figure 6.11: Partial set of screenshots of the LibrarySystem executing prototype obtained after process DM+UCM2UIM.

6.5.2 Generated Prototype after process DM2UIM

This subsection shows the results of the application of DM2UIM and code generation processes to the previously presented Conference Review System domain model.

Based only in the domain model depicted in figure 6.12, and in the rules defined in section 5.2, the prototyped generator outputs a set of XUL and javascript files, and an RDF file. As before, screenshots of the XUL windows rendered by xulrunner, during the prototype execution, are partially shown in figures 6.13 and 6.14.

Figure 6.13 shows the initial window of the generated prototype, and a sequence of windows that appear when the user chooses menu option “Author

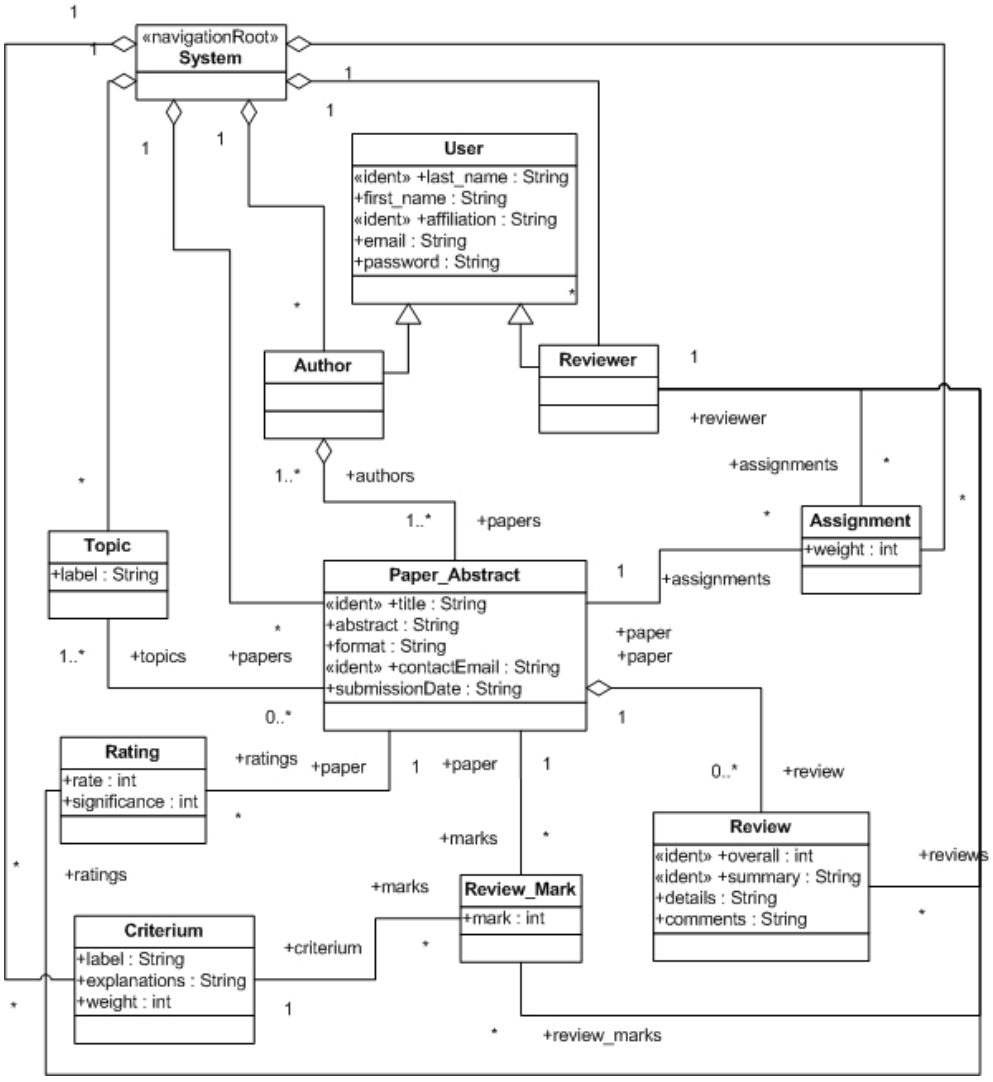

Figure 6.12: Domain model for a Conference Review System (ConferenceReviewSystem).

Collection" and then "Edit Author" (on the left), and when the user chooses menu option "Paper_Abstract Collection" and then "Edit Paper_Abstract" (on the right). Note the expanded list of authors in the Paper_Abstract window, with options "Select Author" and "Unlink Author".

This case study exhibits many-to-many relationships in its domain model. In the generated prototype this is apparent, for instance, in the Paper_Abstract's and in the Author's windows (see fig. 6.13).

Figure 6.14 shows the Paper_Abstract window having the list of related Reviews expanded, with options "Edit Review", "Add Review" and "Remove Re-

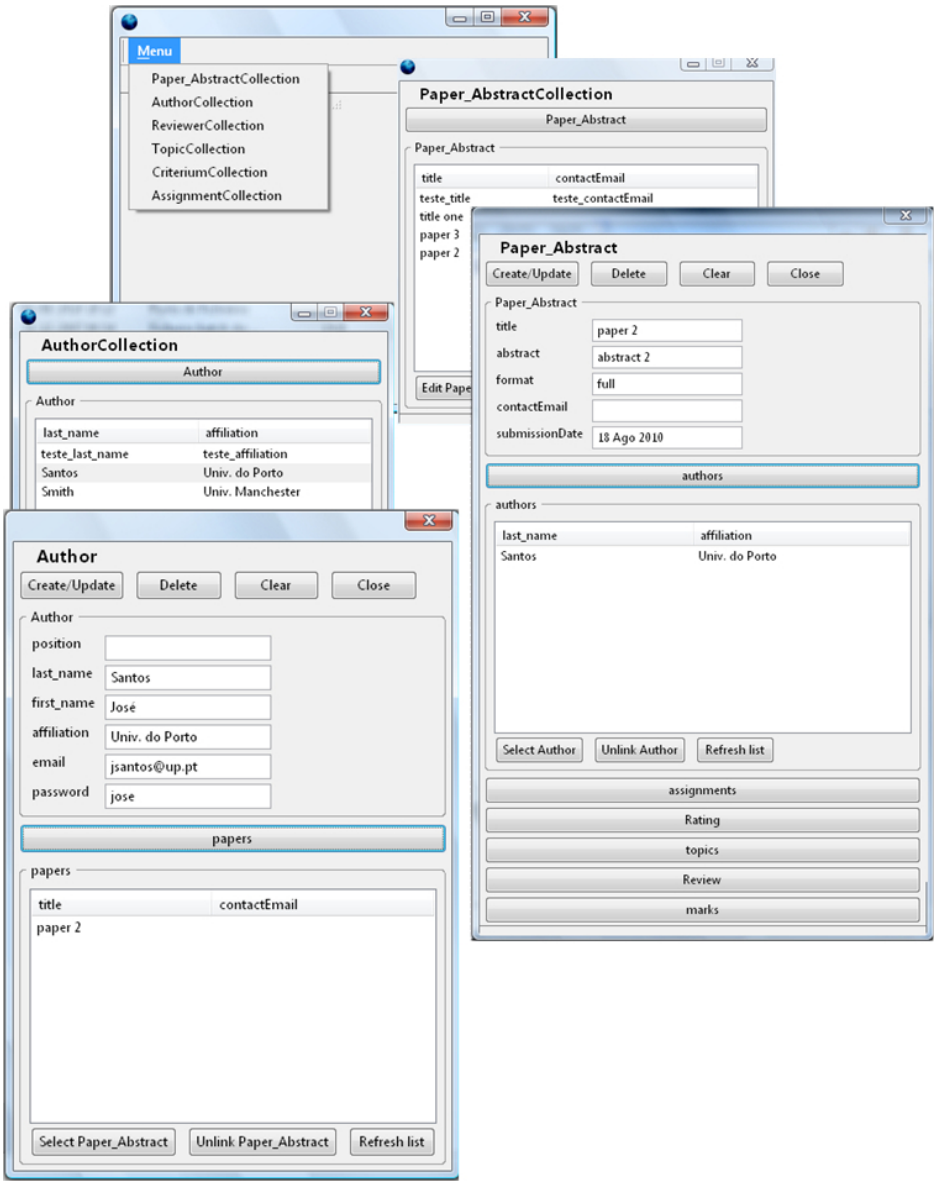

Figure 6.13: Screenshots of the initial window and navitations from menu options "AuthorCollection" (on the left) and "Paper_AbstractCollection" (on the right).

view", followed by the windows for editing a Review and selecting a Reviewer. This is because the relation between *Paper_Abstract* and *Review* is an aggregation relation (refer to the transformation rule in section 5.2.5).

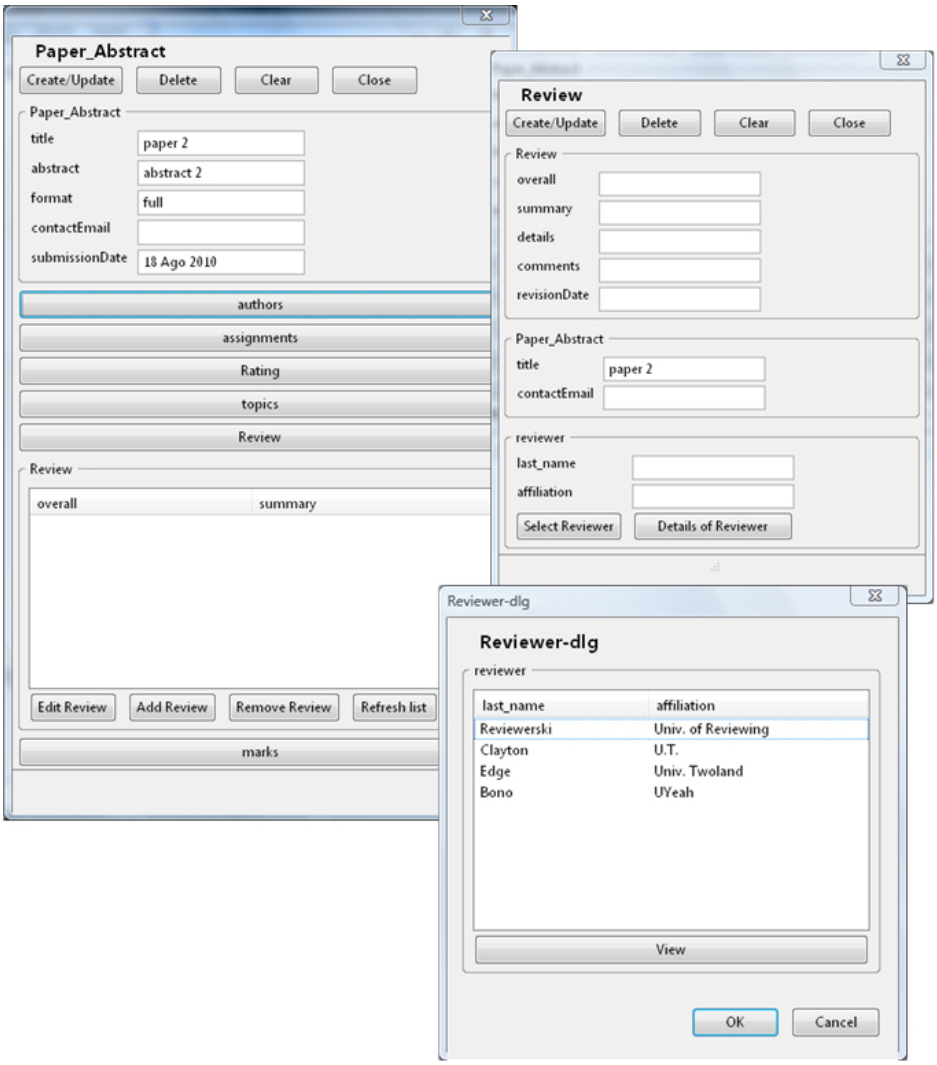

Figure 6.14: Screenshots of the windows flow when the user selects "Edit Review" in the Paper_Abstract window, and then presses "Select Reviewer".

6.5.3 Conference Review System Use Case Model

Figure 6.15 and Table 6.3, respectively show the use case model for the Conference Review System, and the values of the meta-attributes that provide the integration between the DM and the UCM.

There are three actors, or user profiles, in the modeled system, namely *Author*, *Reviewer* and *PCchair*.

The use case model distributes different system functionality by the disparate actors. To keep the case study simple, each actor has only one or two directly related use cases, and some functionality derivable from the DM is not modeled

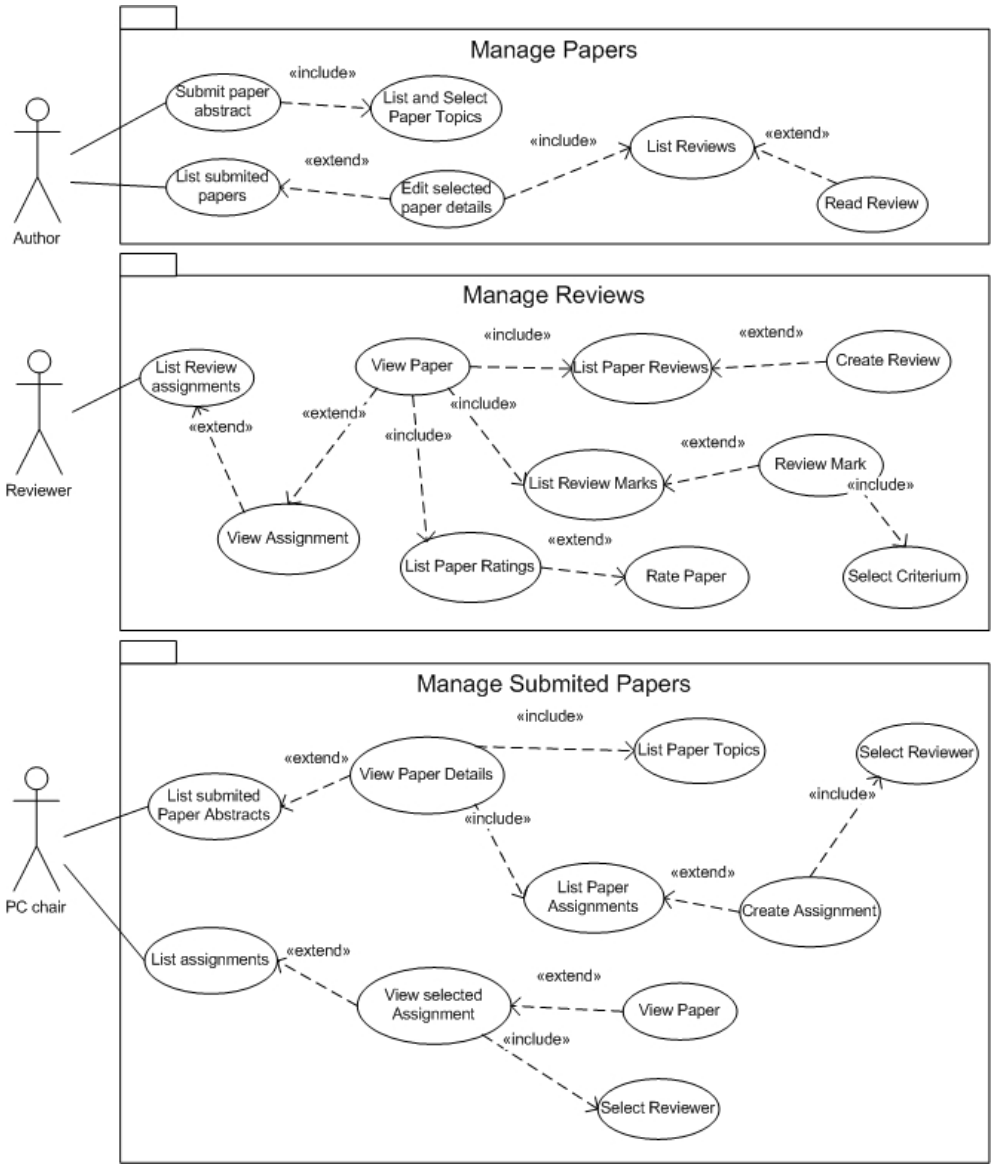

Figure 6.15: Use case model for the Conference Review System.

in the UCM.

6.5.4 Generated Prototype after process DM+UCM2UIM

This subsection shows the results of the application of DM+UCM2UIM and code generation processes to the previously presented Conference Review System domain and use case models.

As shown before, the use of a UCM enables the distribution of the available

Use case	Entity	Associated Operation(s)	Entity Collection
Submit Paper Abstract	Paper_Abstract	Create	False
List submitted Papers	Paper_Abstract		True
Edit Selected Paper Details	Paper_Abstract	Update	False
List Reviews	Review		True
Read Review	Review	Retrieve	False
List Review Assignments	Assignment		True
View Assignment	Assignment	Retrieve	False
View Assignment Paper	Paper_Abstract	Retrieve	False
List Paper Reviews	Review		True
List Review Marks	Review_Mark		True
List Paper Ratings	Rating		True
Create Review	Review	Create	False
Review Mark	Review_Mark	Create	False
Select Criterium	Criterium	Update	True
Rate Paper	Rating	Create	False
List submitted Paper Abstracts	Paper_Abstract		True
View Paper Details	Paper_Abstract	Retrieve	False
List Paper Assignments	Assignment		True
Create Assignment	Assignment	Create	False
Select Reviewer	Reviewer	Update	True
List Paper Topics	Topic		True
List Assignments	Assignment		True
Edit selected Assignment	Assignment	Update	False
View Paper	Paper_Abstract	Retrieve	False

Table 6.3: Entities and operations associated (via tagged values) with some of the use cases in Fig. 6.15.

system functionality by actor. Figure 6.16 shows the initial generated application window, and each different actor's main window and respective menu options.

Figure 6.17 shows two sequences of windows flows, namely one for an author performing use case "List Submited Papers" (on the left), and the other for a PC chair performing use case "List Submited Paper Abstracts" (on-the right). As the constructed prototype does not currently support many-to-many relations when use case driven transformation is being used (process DM+UCM2UIM), the Paper_Abstract windows have been substituted by the respective CAP models. In the figure, it is apparent the difference between the elements generated in the Paper_Abstract interaction spaces when involved in the referred use cases.

For instance, only the Author is able to update the paper abstract, and to list and read the paper's reviews. On the other hand, only the PC chair is able to list and create reviewing assignments to a paper abstract. This conforms to what is modeled in the use case model.

6.6 Discussion of case study's results

The prototyped tool for automating the model to model transformation processes, DM2UIM and DM+UCM2UIM, and the model to code process does not

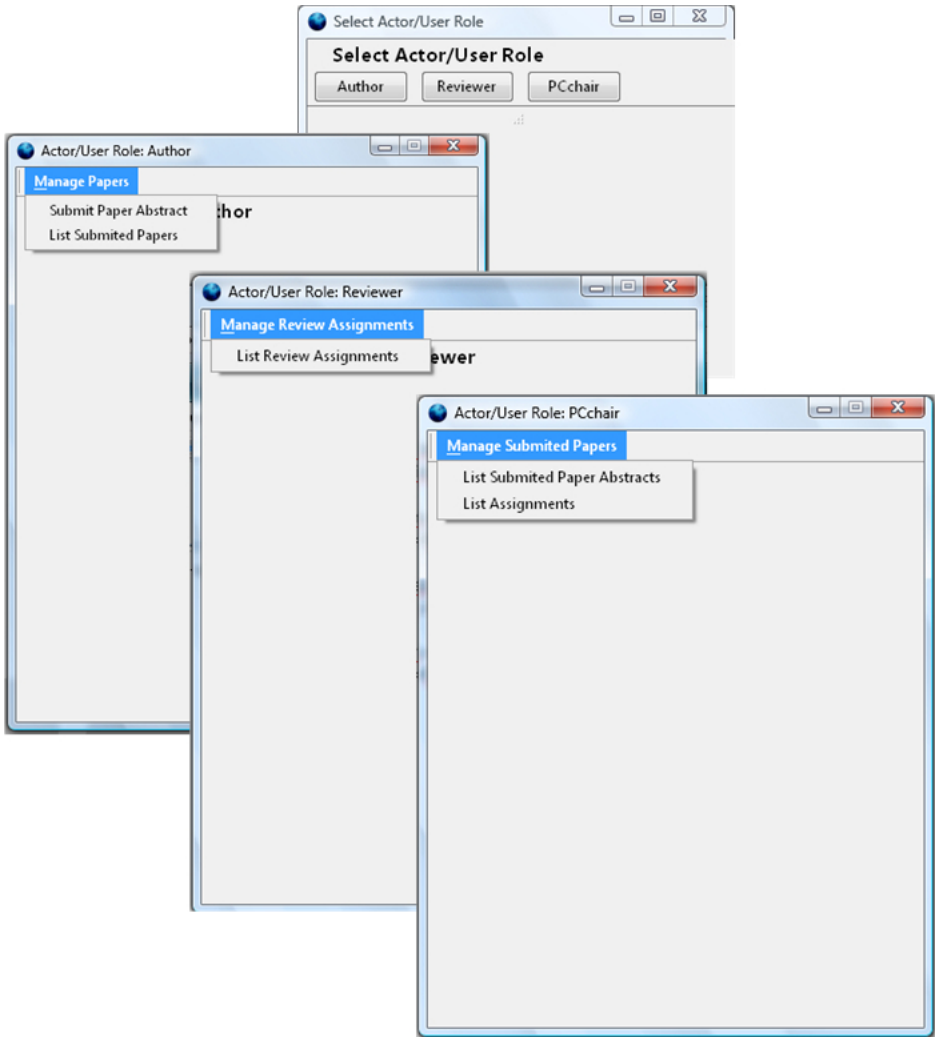

Figure 6.16: Initial window and each actor's main window and respective menu options.

completely support all the defined transformation rules, as seen in the previous subsection. Nevertheless, all the DM2UIM transformation rules and most of the DM+UCM2UIM transformation rules have been implemented. Only the many-to-many relations' transformation rules are not implemented, and this is true only to DM+UCM2UIM process, that is only for use case driven transformation respecting rule DM+UCM2UIM5, when applied to many-to-many associations.

The results of the two presented case studies illustrate the automatic derivation of a UIM from a DM or a DM and a UCM, and the posterior code generation process.

Features generated from base and derived entities have been highlighted, just as features generated from composition and aggregation relations and from simple

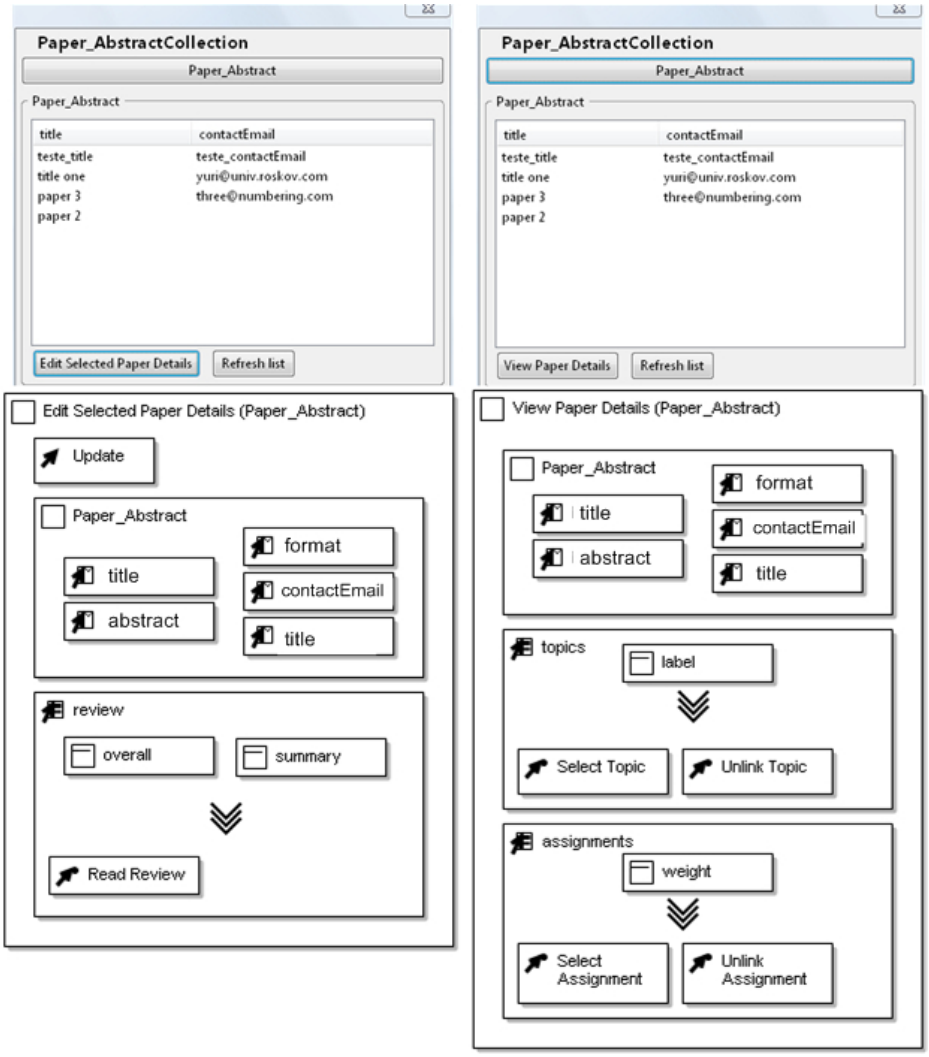

Figure 6.17: Sequences of interaction flows for an author performing use case "List Submited Papers" (on the left), and for a PC chair performing use case "List Submited Paper Abstracts" (on-the right).

associations, including, in the second case study, many-to-many associations.

Features generated from the attributes' data types, triggers, invariants and methods' preconditions have also been verified in the first case study.

The prototyped tool is also able to partially address a "fine grained" use case definition like the one depicted in figure 4.15 (recall section 4.5.3), provided that the including use case and the included ones, that are related through enabling, deactivation or choice relations, all give origin to a unique interaction space. It does not currently support use cases having enabling or deactivating relations involving several interaction spaces, though. This would be the case of use case

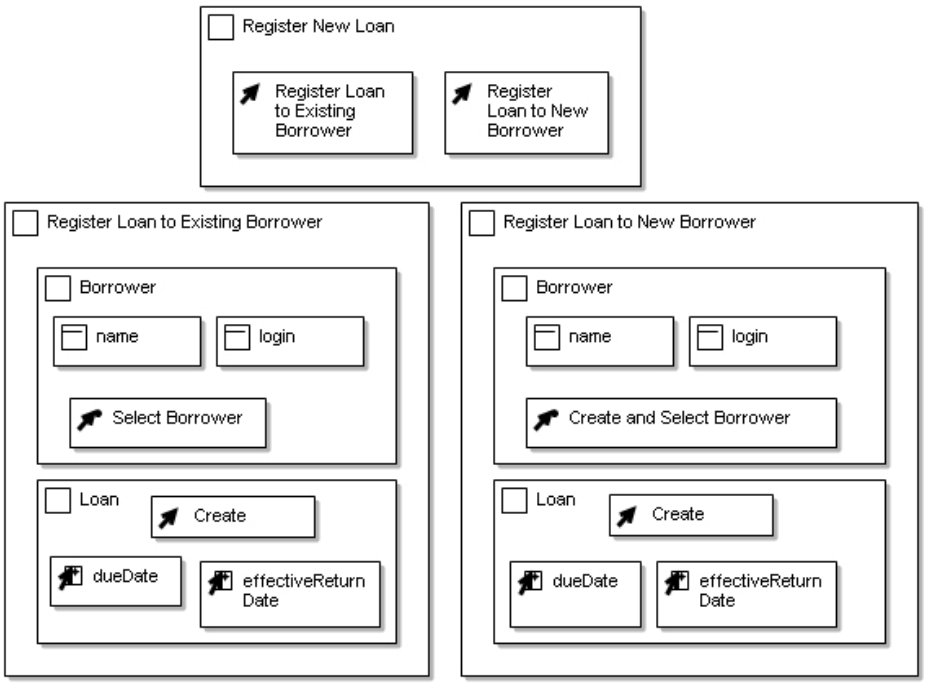

Figure 6.18: CAP of interaction spaces derived from use case "Register New Loan" in figure 4.16.

"Register New Loan" in figure 4.16. Figure 6.18 shows the CAP of interaction spaces derived from use case "Register New Loan" in figure 4.16. The interaction space on the left enables the selection of an existing borrower, by opening an interaction space for selecting one of all available borrowers on the system. The interaction space on the right enables the creation of a new borrower, by opening the borrower creation interaction space.

Note that the CAP is according to the modeled use case, but that it would not be able to create a new loan in the system, because it doesn't allow to select a BookCopy instance, as demanded by the domain model. The inclusion of a use case for selecting a BookCopy instance is lacking in the UCM.

6.7 Assessment of goals satisfaction

In the first chapter of this dissertation, the following research questions have been formulated, based on the conclusions from the state of the art discussion (refer to chapter 3):

1. Is it possible to obtain a UI prototype from a minimal set of model artifacts (a rigorous domain model and a rigorous use case model), without requiring as input a UI model?

2. Can the generated UI prototype take advantage of advanced elements in a domain model:
    - state invariants;
    - operations' pre-/post-conditions;
    - operations rigorously and completely defined by the modeler, that is operations which are completely and formally specified and for which there is only one interpretation;
    - derived attributes;
    - derived classes;
    - triggers or other means of modifying standard CRUD behavior.
3. Can the generated UI prototype take advantage of more flexible elements, in a use case model:
    - typical use case relations (e.g.: inclusion, extension, inheritance);
    - use of constructs typically found in task models for detailing use cases.
4. Can the generated UI be used as a starting point for further refinements towards the UI of the final application?

Besides posing this questions, chapter 1 also declared as research goal the improvement of current approaches to model-driven automatic UI generation, focusing business data-intensive applications with form-based UIs, and we claimed that it would be possible to automatically generate a usable user interface from early, semantically rich, system models, demanding from the modeler less effort than the existing approaches. Furthermore, we believed that it would be possible to generate different user interfaces, depending on the degree of refinement of the information that the user includes in the model, enabling the generation of simple user interfaces from simple early structural models, and complex user interfaces from semantically richer system platform independent models.

This section answers to the previously posed research questions and argues towards the satisfaction of the research goal and the demonstration of the defended thesis.

6.7.1 Is it possible to obtain a UI prototype from a minimal set of model artifacts, without requiring as input a UI model?

It is clear from chapters 4 to 6, that the presented approach allows the generation of a UI prototype based on the rigorous definition of a domain model alone. Indeed, a domain model is the only mandatory model for the UIM and UI prototype automatic generation.

Moreover, it is possible to start by building a DM, generate an executable prototype, use it to validate the model, and then iterate by refining that model or adding an integrated UCM to it.

The automatically generated prototype is obtained through a two step approach in which a UIM is obtained in the first step. This UIM may be modified by the modeler, before generating the final prototype code. The prototype is fully functional (refer to chapter 6), meaning that not only the UI is prototyped but that the system functionality is generated and may be accessed through the generated UI prototype. This allows for the utilization of the prototype regarding the validation of both the UI and the system requirements and modeled structure and functionality. The generated evolutionary prototype, together with the rigorously constructed domain and use case models, and the derived UI model, allows the early validation of the system models and helps in requirements elicitation and validation.

The early UIs generated from early models, may serve for eliciting complex requirements or test the constructed model by executing it through a UI. The UIs generated from semantically richer models may be used for producing the final application UI.

The UI generated can be subject to usability and appearance improvements (without loosing the links to the underlying system model), and can also be used as a basis for subsequent system development.

Further iterations of modeling and prototype usage (with user feedback) shall enable the refinement of the system model and its enrichment with more model elements.

6.7.2 Can the generated UI prototype take advantage of advanced elements in a domain model?

Chapter 5 presented the transformation rules defined for deriving a UIM from the DM alone or from both the DM and the UCM. From the defined rules, one can see that the UIM elements are derived from several features in the DM and UCM, including derived attributes and derived classes (views), among other features.

For enhancing the preciseness of the model, OCL predicates are used to formalize domain class invariants and domain classes operations' preconditions.

The UI generation process takes advantage of the OCL invariants and of the operations' preconditions, defined in the domain model, and of use case model features, to enhance the usability and behavior of the UI.

User defined operations' body is fully specified by making use of an actions language, in order to guarantee the executability of the generated prototype.

Invariant and precondition constraints are ignored in the model-to-model process, because they are not present in the UIM, but are taken into account when generating code from the models. Post-condition constraints are not used, and

they aren't needed because operations are rigorously and completely defined by the modeler through an actions language.

Another advanced feature in the domain model are triggers, which allow the modification of standard CRUD behavior, and may be provided by the modeler through the referred Action-semantics-based actions language (recall section 4.4.3).

6.7.3 Can the generated UI prototype take advantage of more flexible elements, in a use case model?

Chapter 5 also addresses the transformation rules for deriving a UIM from the UCM (integrated with a DM). From the defined rules, one can see that the UIM elements are derived from several features in the UCM, including use case relations, such as inclusion, extension and use case inheritance, and task-model-based relations that provide a way of enabling or deactivating use cases by executing another use case, and of choosing between several use cases for extending another use case.

6.7.4 Can the generated UI be used as a starting point for further refinements towards the UI of the final application?

As defined in the development process proposed in chapter 4, the generated default user interface may be *tunned* by the modeler, or a UI designer, in two points of the process: - After having generated an abstract UI model, but before generating a concrete UI; and, after generating a concrete UI in a XML-based UI description language, such as XUL, that allows for the *a posteriori* customization and application of style sheets.

Also, after having automatically generated the final code, it is possible for a programmer to modify the generated code, and for a UI designer to add or modify the UI skin or style sheets.

This way, it is possible to use the generated UI as a starting point for further refinements towards the final application. This is of great advantage, particularly when one wants to generate UIs for several platforms with little effort. Indeed, although the prototyped proof-of-concept tool generates XUL, Javascript and RDF, a model-to-code transformation tool could be used in the second step of the presented approach (recall Fig. 5.1) to generate code to other target platforms as, for instance, XAML, C# and SQL for Oracle database.

6.8 Comparison with existing approaches

Tables 6.4 to 6.6 add the proposed approach to the MDD approaches surveyed in chapter 3 and compared in section 3.5.4.

		XIS approach	OO-Method	Elkoutbi et al.	Martinez et al.	Forbrig et al.	ZOOM	Our approach
Base classes	Classes	√	√	√	√	√	√	√
	Attributes	√	√	√	√	√	√	√
	Relationships	√	√	√	√	√	√	√
	Class invariants	—	√	—	—	—	√	√
	attributes default values	√	√	—	—	—	√	√
	derived attributes	—	√	—	—	—	—	√
	mandatory attributes	—	√	—	—	—	—	√ (through state invariants)
	user defined operations' syntax	limited (only operation name)	√	—	—	—	√	√
	user defined operations' semantics	—	√	—	—	—	√	√
Lists of values	Enumerated classes	√	—	—	—	—	√	√
Views	Views / derived classes	√ (business entities)	—	—	—	—	—	√
	Mapping to base classes	√	—	—	—	—	—	√
Triggers or other forms of modifying CRUD		—	—	—	—	—	—	√
Operations triggered by state conditions		—	√	—	—	—	√	√

Table 6.4: Our approach vs existing UI generation MDD-approaches fine-grained comparison of the domain model elements.

As can be seen in Fig. 6.4, the approach proposed in this dissertation takes advantage of all the DM elements referred to in the figure.

Fig. 6.5, also shows that the approach proposed in this dissertation takes advantage of all the UCM elements referred to in the figure.

		XIS approach	OO-Method	Elkoutbi et al.	Martinez et al.	Forbrig et al.	ZOOM	Our approach
Actors (user roles)		√	User roles represented as classes in the object model.	√	√	√	—	√
Use cases		√	—	√	√	√	—	√
Use case extension		√	—	√	√	√	—	√
Use case inclusion		√	—	√	√	√	—	√
Mapping to base classes	Target views / operations	√	—	—	—	—	—	√
	behaviour	—	—	Sequence diagrams	Collaboration diagrams	Task model	—	Included low-level use cases and Task-model-like relations

Table 6.5: Our approach vs existing UI generation MDD-approaches fine-grained comparison of the use case model elements.

Table 6.6 shows a feature comparison between the proposed approach and the ones surveyed in chapter 3.

As mentioned in chapter 3, 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 DM. The difference is that, in our approach, relations between entities are inferred from the DM, 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.

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

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.

Our approach, thus, puts together the advantages of the surveyed approaches and adds a few features of its own, namely:

- It makes possible to generate an application prototype from an evolving,

	XIS approach	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	generates only UI state model	requires	generates/ allows configuration
Is able to generate a UIM/UIP from non-UI system models	√ (in smart approach)	(only from domain model)	—	√ (non funtional 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 6.6: Feature comparison between the current approaches and the proposed approach

possibly incomplete regarding user requirements, but rigorous and integrated, domain model and optionally a use case model;

- It generates a concrete UI, following a model-driven paradigm, but does not demand the modeler's construction of a UIM. Instead, it derives a default UIM from the domain and use case models, and allows the modeler to optionally modify the generated UIM, before automatically generating the final code;

- It makes use of derived attributes and derived entities (views), in the DM, to better specify "boundary" entities;

- It takes 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;
- It makes 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;
- It allows 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;
- It derives a default use case model from a domain model, easing the process of developing a use case model integrated with the system domain model.

6.9 Conclusions

This chapter presented the proof-of-concept tool that has been developed for validating the proposed approach for data-intensive forms-based interactive applications development.

Through two case studies the feasibility of the approach has been demonstrated. Also, it is possible to generate a UIM and interactive running prototype by developing only a DM, which is a low-effort approach to executable evolutionary prototyping that enables the use of the prototype by users and other stakeholders. When going deep into modeling, the modeler will have, of course, much more effort, depending on what he/she wants to put in the model.

It has also been confirmed, in this chapter, the fulfillment of the proposed research goals.

Chapter 7

Conclusions

This chapter summarizes the results obtained, presents conclusions and points out open issues for future research.

7.1 Results and contributions to the state of art

This Ph.D. dissertation discussed the development of an approach for the model-driven automatic generation of fully functional (executable) interactive applications, from early system models, with minimum effort. Its feasibility has been validated through two case studies, and the research results have been published in four conference papers and one doctoral symposium. The research contributions to the state-of-art, including published results, have been addressed in section 1.6.

We have focused our research on the model-to-model transformation from DM to UIM and from DM+UCM to UIM. Indeed, when regarding the state-of-art approaches, this is where our contribution can be more relevant. In fact, we believe that approaches such as the XIS approach or the Wisdom approach could benefit from this research's results, hence providing a default UIM to the modeler that he/she could then refine and modify according to the system requirements.

In section 6.7, the research questions have been assessed, and all had a confirmatory answer. The comparison to the state-of-art approaches, has been accomplished in section 6.8.

In what respects the effort needed to automatically generate a usable user interface from early, semantically rich, system models, we can divide the conclusion into three points (refer to sections 3.5, 6.7 and 6.8):

- When the goal is to obtain a simple executable prototype, which maps the data structures and provides CRUD operations on those structures: The proposed approach indeed demands less effort from the modeler to build the necessary models. In fact, a domain classes model suffices to get a fully executable prototype, which may be precious in attaining users feedback.

- When the goal is to obtain a more refined prototype, which may imply the definition of views, invariant constraints, operations' preconditions and respective body specification, and triggers: The proposed approach demands from the modeler the knowledge of OCL and an action semantics language. Provided that knowledge, the approach demands at most the same effort as the OO-Method, as a domain model plus specifications is what is needed to get a fully executable prototype.

- When the goal is to obtain an even more refined prototype, that must be able to separate functionality by actor, or even provide some functionality that cannot by fully derived from the domain model: The effort demanded by the proposed approach can only be comparable to the effort demanded by the XIS approach, as the other ones either don't support a use case driven approach, or demand the full construction of a UIM. And, it demands less effort than the XIS smart approach, because for specifying the same things as the XIS Entities View, it only needs a domain model, while XIS needs a Domain View plus a BusinessEntities View, and for specifying the same things as the XIS Use-Cases View, it only needs a use case model, while XIS needs a UseCases View plus an Actors View. Further, the XIS approach demands the stereotyping of every model element.

  Having said that, it lacks to say that the proposed approach DM and UCM enable a richer model than the one provided by the XIS approach, as the DM allows the definition of constraints, user defined operations and triggers, and the UCM allows the usage of use case relations, either UML-based of task-model-inspired.

To obtain an intermediate model, before its transformation into code, like the two step approach prescribed (see section 4.2), is a typical model-driven solution. The prototyped tool has the M2M and M2C transformation rules hard-coded in C# classes. A more flexible solution would be to use an MDA tool, like AtlanMod [1], for the M2M transformation process, and templates, like in the XIS approach, for the M2C transformation process. This was not the focus of this research, though.

We believe that when MDD is in a more mature state, model transformations will be handled by available tools, similar to today's compilers. By that time, the modelers will only have to handle with models construction, trending to specialize themselves in some domain areas, and searching for M2M and M2C transformation tools in the market. This way, the construction of such transformation tools is left to specialized companies, and system modeling will be within reach of business domain experts.

[1] http://www.emn.fr/z-info/atlanmod/index.php/Main_Page

7.2 Future work

Concerning the problem of automatically generating a UIM and a fully executable interactive prototype, the approach presented in this dissertation is an answer that demands little effort from the modeler, at least in the first iterations, towards constructing an interactive system model. Some open research related problems remain, though. The most relevant ones are described below:

- Our proposal allows for the modeler to modify the generated UIM and even the generated final code, which being an iterative process, brings some inherent problems, such as how to handle a new UIM generation when the modeler modifies the previously generated UIM, or how to handle final code generation when the previously generated code has been modified. A separation mechanism that can discern the generated parts of UIM or code from the parts that are manually added or modified by the user, would contribute to this problem's mitigation. These are issues of difficult solution, and were not addressed in this research.

- Related to the previous point, are aspects of integration of the generated prototype with previously existing code or complete applications. The allowance of non-human actors in our UCM, and the possibility to define the integration interface are also open issues for future research.

- For a more broader experimentation, within an industrial context, a better tool support is needed. The import from UML or XMI, produced by UML diagraming tools, and the development of code-generators for other target platforms, ideally based on existing technologies and standards, are possible evolutions for practical tools implementation.

- The presented declarative transformation rules have been implemented imperatively in a proof-of-concept tool. Future work could build a transformation engine that can interpret the declarative rules, hence promoting the separation between the transformation engine and the transformation rules, which would enable the rules customization. This evolution would need the identification of a rules definition metamodel that would allow the definition of the proposed rules and also of new rules. Ideally, this could be based on existing technology and standards, such as QVT.

- One distinctive aspect of our approach is that it takes advantage of declarative constraints (state invariants and operations' preconditions) to validate data entered into forms. An interesting research direction would be to try to take advantage of those declarative constraints to suggest values to the user, or to limit the values that the user can select to a given form field.

- The approach presented focused forms-based data-intensive business applications. Even in the context of business applications, some non-forms based user interfaces could be advantageous. For instance tree or graph manipulation interfaces, when working with some data structures, or even multimedia interfaces, such as voice interfaces, or virtual reality interfaces. The model features from which these could be automatically generated is a research topic. Another research issue has to do with UI designer know-how, and how, or to what point, can it be embedded into an UI automatic generation approach.

Bibliography

[AD06] Samir Ammour and Philippe Desfray. A concern-based technique for architecture modelling using the uml package merge. *Electr. Notes Theor. Comput. Sci.*, 163(1):7–18, 2006.

[APB+99] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, and Jonathan E. Shuster. UIML: An appliance-independent XML user interface language. *http://www8.org/w8-papers/5b-hypertext-media/uiml/uiml.html*, 1999. [Visited in 2007-01-10].

[Art04a] Artima. Contract-driven Development - A conversation with Bertrand Meyer, Part III - by Bill Venners, 2004. www.artima.com/intv/contestP.html.

[Art04b] Artima. Design by Contract - A conversation with Bertrand Meyer, Part II - by Bill Venners, December 2004. www.artima.com/intv/contractsP.html.

[AvEPvW04] P. Achten, M. van Eekelen, R. Plasmeijer, and A. van Weelden. Automatic generation of editors for higher-order data structures. *Programming Languages and Systems. Second Asian Symposium, APLAS 2004. Proceedings (Lecture Notes in Computer Science Vol.3302)*, pages 262 – 79, 2004.

[BBF+87] Bill Betts, David Burlingame, Gerhard Fischer, Jim Foley, Mark Green, David Kasik, Stephen T Kerr, Dan Olsen, and James Thomas. Goals and objectives for user interface software. *SIGGRAPH Comput. Graph.*, 21(2):73–78, 1987.

[Boo93] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Professional, 2nd edition edition, 1993.

[CL99] Larry Constantine and Lucy Lockwood. *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. Addison-Wesley Professional, 1999.

[CN04] Pedro F. Campos and Nuno Jardim Nunes. A uml-based tool for designing user interfaces. In *UML Satellite Activities*, pages 273–276, 2004.

[Con95] Larry Constantine. What do users want ? Engineering usability into software. Reprinted and revised (June 2000) from Windows Tech Journal, Dec. 1995. Available in http://www.foruse.com/articles/whatusers.pdf, 1995.

[Con03] Larry L. Constantine. Canonical abstract prototypes for abstract visual and interaction design. In J. Falcão e Cunha J.A. Jorge, N. Jardim Nunes, editor, *Proceedings of the DSV-IS 2003, LNCS 2844*, number 2844 in LNCS, pages 1 – 15. Springer-Verlag Berlin Heidelberg, 2003.

[Con06a] Larry Constantine. Activity modeling: Toward a pragmatic integration of activity theory with usage-centered design. Thecnical Paper. Revision 2.0, 2006.

[Con06b] Larry Constantine. *The Persona Lifecycle*, chapter Users, Roles, and Personas, page chapter 8. Morgan-Kaufmann, 2006.

[CWNL03] Larry Constantine, Helmut Windl, James Noble, and Lucy Lockwood. From abstraction to realization: Canonical abstract prototypes for user interface design. Revised Working Paper. Available in http://www.foruse.com/articles/canonical.pdf, July 2003.

[dCF08] António Miguel Rosado da Cruz and João Pascoal Faria. Automatic generation of interactive prototypes for domain model validation. In José Cordeiro, Boris Shishkov, Alpesh Ranchordas, and Markus Helfert, editors, *Proceedings of the International Conference on Software Engineering and Data Technologies (ICSoft 2008)*, volume SE/GSDCA/MUSE, pages 206–213. INSTICC - Institute for Systems and Technologies of Information, Control and Communication, INSTICC Press, July 2008.

[dCF09] António Miguel Rosado da Cruz and João Pascoal Faria. Automatic generation of user interface models and prototypes from domain and use case models. In Boris Shishkov, José Cordeiro, and Alpesh Ranchordas, editors, *Proceedings of the International Conference on Software Engineering and Data Technologies (ICSoft 2008)*, volume 1, pages 169–176. INSTICC - Institute for Systems and Technologies of Information, Control and Communication, INSTICC Press, July 2009.

[dCF10] António Miguel Rosado da Cruz and João Pascoal Faria. A metamodel-based approach for automatic user interface generation. In *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering (Models 2010), Oslo, Norway, October 2010 (to appear)*, 2010.

[Dem79] Tom Demarco. *Structured Analysis and System Specification*. Prentice Hall, 1979.

[DFAB98] Alan Dix, Janet Finlay, Gregory Abowd, and Russell Beale. *Human-Computer Interaction*. Prentice Hall, 2nd edition edition, 1998.

[DLW05] Dirk Draheim, Christof Lutteroth, and Gerald Weber. Generative programming for C#. *ACM SIGPLAN Notices*, 40(8):29–33, August 2005.

[DS90] Prasun Dewan and Marvin Solomon. An approach to support automatic generation of user interfaces. *ACM Transactions on Programming languages and Systems*, 12(4):566–609, October 1990.

[dSSdS08] João de Sousa Saraiva and Alberto Rodrigues da Silva. The ProjectIT-Studio UMLModeler: A tool for the design and transformation of UML models. In *Proceedings 3ª Conferencia Ibérica de Sistemas y Tecnologías de la Información (CISTI 2008)*, Campus de Ourense, Ourense, Spain, 2008. Universidad de Vigo.

[dSSSM07] Alberto Rodrigues da Silva, João Saraiva, Rui Silva, and Carlos Martins. XIS - UML profile for extreme modeling interactive systems. In *4th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2007)*. IEEE Computer Society, March 2007.

[EKK06] M. Elkoutbi, I. Khriss, and R.K. Keller. Automated prototyping of user interfaces based on UML scenarios. *Journal of Automated Software Engineering*, 13(1):5–40, January 2006.

[Far07] João Pascoal Faria. Model driven development (MDD) using formal methods and UML. Presentation at Seminário de Sistemas de Informação, Escola Superior de Tecnologia e Gestão - IPVC, Viana do Castelo, March 2007. (in Portuguese).

[FDRS04] Peter Forbrig, Anke Dittmar, Daniel Reichart, and Daniel Sinnig. From models to interactive systems tool support and XIML. In Hallvard Trætteberg, Pedro J. Molina, and Nuno Jardim Nunes,

editors, *Proceedings of the First International Workshop on Making model-based user interface design practical: usable and open methods and tools (MBUI 2004)*, volume 103 of *CEUR Workshop Proceedings*, Funchal, Madeira, Portugal, January 2004. Available at http://ceur-ws.org.

[Fer05] Luis G. M. Ferreira. Formalizing markup languages for user interface. Master's thesis, Escola de Engenharia, Universidade do Minho, Braga, 2005.

[FL98] John Fitzgerald and Peter Larsen. *Modelling Systems. Practical Tools and Techniques in Software Development.* Cambridge University Press, 1998.

[Fra03] David S. Frankel. *Model Driven Architecture - Applying MDA to Enterprise Computing.* Wiley Publishing, Inc., Indianapolis, Indiana, 2003.

[Gre04] Shirley Gregor. The struggle towards an understanding of theory in information systems. *Information systems foundations: Constructing and Criticising Workshop*, July 2004. Available online at http://epress.anu.edu.au/info_systems/mobile_devices/ch01.html. The Australian National University.

[Ins06] European Software Institute. A presentation of MDD basics: Model-driven development (MDD) tutorial for managers. ModelWare - ReMOSitory, available at http://www.modelware-ist.org/, September 2006. European Software Institute, Corporación Tecnológica Tecnalia.

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software development Process.* Addison-Wesley, 1999.

[JCJv92] Ivar Jacobson, Magnus Christerson, Patrick Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering - A Use Case Driven Approach.* ACM Press / Addison-Wesley, 1992.

[JSL+05] Xiaoping Jia, Adam Steele, Hongming Liu, Lizhang Qin, and Chris Jones. Using ZOOM approach to support MDD. In *Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP'05)*, Las Vegas, Nevada, USA, June 27-30 2005.

[JSQ+07] Xiaoping Jia, Adam Steele, Lizhang Qin, Hongming Liu, and Chris Jones. Executable visual software modeling–the ZOOM approach. *Software Quality Control*, 15(1):27–51, 2007.

[JSS+07] Homa Javahery, Daniel Sinnig, Ahmed Seffah, Peter Forbrig, and T. Radhakrishnan. *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, 2007.

[Jéz05] Jean-Marc Jézéquel. Model transformation techniques, 2005. (available in http://modelware.inria.fr/static_pages/slides/ModelTransfo.pdf).

[Kah01] James P. Kahan. Basic vs. applied research: The wrong dichotomy? RAND Europe, Oslo, Norway, October 2001.

[KEK01] Ismaïl Khriss, Mohammed Elkoutbi, and Rudolf K. Keller. Automatic synthesis of behavioral object specifications from scenarios. *J. Integr. Des. Process Sci.*, 5(3):53–77, 2001.

[KLM03] E. Kantorowitz, A. Lyakas, and A. Myasqobsky. A use case-oriented user interface framework. *Proceedings IEEE International Conference on Software - Science, Technology and Engineering (SwSTE'03)*, pages 93 – 100, 2003.

[Knu92] E. Donald Knuth. *Literate Programming*. University of Chicago Press, Chicago, 1992.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Press, 2008.

[Lan96] James A. Landay. SILK: sketching interfaces like krazy. In *CHI '96: Conference companion on Human factors in computing systems*, pages 398–399, New York, NY, USA, 1996. ACM.

[LLK04] Patrick Lay and Stefan Lüttringhaus-Kappel. Transforming XML schemas into Java Swing GUIs. In Peter Dadam and Manfred Reichert, editors, *GI Jahrestagung (1), INFORMATIK 2004 - Informatik verbindet, Band 1, Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 20. September - 24. September 2004 in Ulm*, volume P-50 of *LNI*, pages 271–276. GI, 2004.

[LM05] María Lázaro and Esperanza Marcos. Research in software engineering: Paradigms and methods. In Jaelson Castro and Ernest Teniente, editors, *CAiSE Workshops (2)*, pages 517–522. FEUP Edições, Porto, 2005.

[LM06] María Lázaro and Esperanza Marcos. An approach to the integration of qualitative and quantitative research methods in software engineering research. In Esperanza Marcos, Mark Lycett, César J. Acuña, and Juan M. Vara, editors, *PhiSE*, volume 240 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.

[LNHL02] James Lin, Mark W. Newman, Jason I. Hong, and James A. Landay. Denim: An informal sketch-based tool for early stage web design. In *Proceedings of AAAI 2002 Spring Symposium (Sketch Understanding Workshop)*, Stanford, CA, 2002.

[Mar02] Antti Martikainen. An XML-based framework for developing usable and reusable user interfaces for multi-channel applications. Pro gradu thesis, report, Department of Computer Science, University of Helsinki, May 2002.

[Mar07] Carlos Alberto Rodrigues Martins. Modelação de interfaces gráficas no ambito do ProjectIT. Master's thesis, University of Madeira, Funchal, Portugal, 2007. MSc thesis in portuguese.

[MAT01] A. Mahfoudhi, M. Abed, and D. Tabary. From the formal specifications of users tasks to the automatic generation of the HCI specifications. *People and Computers XV - Interaction without Frontiers. Joint Proceedings of HCI 2001 and IHM 2001*, pages 331 – 47, 2001.

[MESP02] A. Martinez, H. Estrada, J. Sánchez, and O. Pastor. From early requirements to user interface prototyping: A methodological approach. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE 2002)*, pages 257–260, 2002.

[Mey06] Bertrand Meyer. *Dependable Systems: Software, Computing, Networks*, volume 4028 of *Lecture Notes in Computer Science*, chapter Dependable Software, pages 1–33. Springer Berlin / Heidelberg, 2006.

[MH03] Pedro J. Molina and Javier Hernández. Just-UI: Using patterns as concepts for IU specification and code generation. In *Perspectives on HCI Patterns: Concepts and Tools (CHI'2003 Workshop)*, 2003.

[Mol04] Pedro J. Molina. User interface generation with olivanova model execution system. In *IUI '04: Proceedings of the 9th international conference on Intelligent user interfaces*, pages 358–359, New York, NY, USA, 2004. ACM.

[Mor03] Pedro Juan Molina Moreno. *Especificación de interfaz de usuário: De los requisitos a la generación automática*. PhD thesis, Universidad Politécnica de Valencia, Marzo 2003. (In spanish).

[Moz] Mozilla. XML user interface language (XUL) project. http://www.mozilla.org/projects/xul/. [Visited in 2008-09-19].

[MPM+01] P.J. Molina, O. Pastor, S. Marti, J.J. Fons, and E. Insfram. Specifying conceptual interface patterns in an object-oriented method with automatic code generation. *Proceedings Second International Workshop on User Interfaces in Data Intensive Systems. UIDIS 2001*, pages 72 – 9, 2001.

[MR92] Brad A. Myers and Mary Beth Rosson. Survey on user interface programming. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 195–202, New York, NY, USA, 1992. ACM Press.

[MSDa] MSDN. Visual studio developer center. http://msdn.microsoft.com/en-us/vstudio. [Visited in 2008-09-19].

[MSDb] MSDN. XAML overview. http://msdn2.microsoft.com/en-us/library/ms752059.aspx. [visited in 2007-01-11].

[NC06] Paul Nguyen and Robert Chun. Model driven development with interactive use cases and UML models. In *Software Engineering Research & Practice Conference (SERP 2006)*, Las Vegas, Nevada, June 26-29 2006.

[Nun01] Nuno Jardim Nunes. *Object Modeling for User-Centered Development and User Interface Design: The Wisdom Approach*. PhD thesis, University of Madeira, July 2001.

[Oli09] Eugénio Oliveira. Mic - metodologias de investigação científica (apresentação de apoio à disciplina de mic). Available in http://paginas.fe.up.pt/ eol/PRODEI/mic0910_files/Teorias.pdf (in portuguese), October 2009.

[OMG01] OMG. Omg unified modeling language specification (action semantics), December 2001. Action Semantics Draft Adopted Specification. Base Document: UML 1.4.

[OMG03] OMG. Uml 2.0 ocl specification, October 2003.

[OMG09a] OMG. Unified Modeling Language (OMG UML), Infrastructure, February 2009.

[OMG09b] OMG. Unified Modeling Language (OMG UML), Superstructure, February 2009.

[Ope04] OASIS Open. User Interface Markup Language (UIML) Specification. *http://www.oasis-open.org/committees/documents.php?wg_abbrev=uiml.*, 2004. editors: Marc Abrams and James Helms.

[Out] OutSystems. Outsystems agile platform - technical overview. Published in http://www.outsystems.com. [Visited in 2008-09-20].

[Pai06] Ana Cristina Paiva. *Automated Specification-Based Testing of Graphical User Interfaces*. PhD thesis, Faculty of Engineering, University of Porto, 2006.

[Pat03] F. Paternò. *The handbook of Task Analysis for Human-Computer Interaction*, chapter ConcurTaskTrees: An engineered notation for task models, pages 483–503. D. Diaper and N. Stanton, 2003.

[PB08] Ilia Petrov and Alejandro Buchmann. Architecture of omg mof-based repository systems. In *Proceedings of iiWAS2008*, 2008.

[PE99] Angel Puerta and Jacob Eisenstein. Towards a general computational framework for model-based interface development systems. In *IUI '99: Proceedings of the 4th international conference on Intelligent user interfaces*, pages 171–178, New York, NY, USA, 1999. ACM Press.

[PE01] Angel Puerta and Jacob Eisenstein. XIML: A universal language for user interfaces. Technical report, RedWhale Software, 2001.

[PI03] Oscar Pastor and Emilio Insfrán. OO-Method, the methodological support for Oliva Nova model execution system. Technical report, Care Technologies, 2003. White paper. Available at http://www.care-t.com.

[Pin00] Paulo Pinheiro da Silva. 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*, volume 1946 of *Lecture Notes in Computer Science*, pages 207–226, Limerick, Ireland, 2000. Springer Berlin / Heidelberg.

[Pin02] Paulo Pinheiro da Silva. *Object Modelling of Interactive Systems: The UML$_i$ Approach*. PhD thesis, Faculty of Science and Engineering, University of Manchester, 2002.

[PIP+97] Oscar Pastor, Emilio Insfrán, Vicente Pelechano, José Romero, and José Merseguer. 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, 1997. Springer-Verlag.

[PMI04] Oscar Pastor, Juan Carlos Molina, and Emilio Iborra. 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.

[PMM97] Fabio Paternò, Cristiano Mancini, and Silvia Meniconi. Concurtasktrees: A diagrammatic notation for specifying task models. In *INTERACT '97: Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction*, pages 362–369, London, UK, UK, 1997. Chapman & Hall, Ltd.

[PP01] Paulo Pinheiro da Silva and Norman W. Paton. User Interface Modelling with UML. In H. Jaakkola, H. Kangassalo, and E. Kawaguchi, editors, *Information Modelling and Knowledge Bases XII*, pages 203–217, Amsterdam, The Netherlands, 2001. IOS Press. 10th European-Japanese Conference on Information Modelling and Knowledge Representation, May 2000, Saariselkä, Finland).

[Pre05] Roger Pressman. *Software Engineering: A Practitioner's Approach*. McGraw Hill, 6th edition edition, 2005.

[RBL+90] James R. Rumbaugh, Michael R. Blaha, William Lorensen, Frederick Eddy, and William Premerlani. *Object-Oriented Modeling and Design*. Prentice-Hall, 1st edition edition, 1990.

[RFD04] Daniel Reichart, Peter Forbrig, and Anke Dittmar. Task models as basis for requirements engineering and software execution. In *TAMODIA*, pages 51–58, 2004.

[RFSS07] Frank Radeke, Peter Forbrig, Ahmed Seffah, and Daniel Sinnig. PIM Tool: Support for pattern-driven and model-based UI development. In *Task Models and Diagrams for Users Interface Design (TAMODIA 2006)*, volume 4385/2007 of *Lecture Notes in Computer Science*, pages 82–96. Springer Berlin / Heidelberg, 2007.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language reference manual*. Addison-Wesley, 1999.

[SDJ07] Dag I. K. Sjoberg, Tore Dyba, and Magne Jorgensen. The future of empirical methods in software engineering research. In *FOSE '07: 2007 Future of Software Engineering*, pages 358–378, Washington, DC, USA, 2007. IEEE Computer Society.

[Sil03] Alberto Silva. The XIS approach and principles. In IEEE Computer Society, editor, *Proceedings of the 29th EUROMICRO Conference "New Waves in System Architecture" (EUROMICRO '03)*, 2003.

[Sil04] Alberto Silva. O programa de investigação "ProjectIT". Technical report, v1.0, INESC-ID, October 2004. In portuguese. Available in http://isg.inesc-id.pt/alb/uploads/1/193/pit-white-paper-v1.0.pdf.

[Som07] Ian Sommerville. *Software Engineering*. Addison-Wesley, 8th edition, 2007.

[SSC+96] Pedro A. Szekely, Piyawadee Noi Sukaviriya, Pablo Castells, Jeyakumar Muthukumarasamy, and Ewald Salcher. Declarative interface models for user interface construction tools: the mastermind approach. In *Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction*, pages 120–150, London, UK, UK, 1996. Chapman & Hall, Ltd.

[SSFV07] Alberto Silva, João Saraiva, David Ferreira, and Carlos Videira. Integration of RE and MDE paradigms: the ProjectIT approach and tools. *IET Software*, 1(6):294–314, 2007.

[Sun] Sun. Netbeans IDE - features. http://www.netbeans.org/features/. [Visited in 2008-09-19].

[SV05] Alberto Silva and Carlos Videira. *UML, Metodologias e Ferramentas CASE*, volume volume 1. Centro Atlântico, Lda., 2nd edition, 2005. (In portuguese).

[SV08] Alberto Silva and Carlos Videira. *UML, Metodologias e Ferramentas CASE*, volume volume 2. Centro Atlântico, Lda., 2nd edition, 2008. (In portuguese).

[SVS+06] Alberto Silva, Carlos Videira, João Saraiva, David Ferreira, and Rui Silva. The ProjectIT-Studio, an integrated environment for the development of information systems. *Second International Conference on Innovative Views of .Net Technologies - IVNET*, 2006.

[SWM06] Bernd Schoeller, Tobias Widmer, and Bertrand Meyer. Making specifications complete through models. In Ralf Reussner, Judith Stafford, and Clemens Szyperski, editors, *Architecting Systems with Trustworthy Components*, volume 3938 of *Lecture Notes in Computer Science*, pages 48–70. Springer Berlin / Heidelberg, 2006.

[Træ02] Hallvard Trætteberg. *Model-based User Interface Design*. PhD thesis, Information Systems Group, Department of Computer and Information Sciences, Faculty of Information Technology, Mathematics and Electrical Engineering. Norwegian University of Science and Technology, May 2002.

[WBP+03] Jos Warmer, Wim Bast, Diane Pinkley, Mario Herrera, and Anneke Kleppe. *MDA Explained - The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, 2003.

[WFDR05] Andreas Wolff, Peter Forbrig, Anke Dittmar, and Daniel Reichart. Linking GUI elements to tasks: supporting an evolutionary design process. In *TAMODIA '05: Proceedings of the 4th international workshop on Task models and diagrams*, pages 27–34, New York, NY, USA, 2005. ACM.

[WFR05] Andreas Wolff, Peter Forbrig, and Daniel Reichart. Tool support for model-based generation of advanced user-interfaces. In Andreas Pleuss, Jan Van den Bergh, Heinrich Hussmann, and Stefan Sauer, editors, *Proceedings of the MoDELS'05 Workshop on Model Driven Development of Advanced User Interfaces*, Montego Bay, Jamaica, October 2005.

[YJ02] Joseph W. Yoder and Ralph E. Johnson. The adaptive object-model architectural style. In *WICSA 3: Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture*, pages 3–27, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[You88] Edward Yourdon. *Modern Structured Analysis*. Prentice Hall, 1988.