From i* Requirements Models to Conceptual Models of a Model Driven Development Process

Fernanda Alencar[1,2], Beatriz Marín[1], Giovanni Giachetti[1], Oscar Pastor[1], Jaelson Castro[2], João Henrique Pimentel[2]

[1] Universidad Politécnica de Valencia, Camino de Vera s/n, CP:46022, Valencia, Spain
{fribeiro, bmarin, ggiachetti, opastor}@dsic.upv.es,
[2] Universidade Federal de Pernambuco, Av. Prof. Luiz Freire s/n, 50740-540, Recife, Brazil
fmra@ufpe.br, {jbc, jhcp}@cin.ufpe.br

Abstract. A good understanding of the systems requirements has a high impact in the successful development of software products. Therefore, an appropriate requirements model must provide a comprehensive structure for what must be elicited, evaluated, specified, consolidated, and modified, instead of just providing facilities for software specifications. Since there is a well-known gap between requirements specifications and final software products, we propose the integration of Goal-Oriented Requirements Engineering (GORE) and Model-Driven Development (MDD) to solve this gap. The core of our proposal is comprised by a set of guidelines to automate the process of going from an initial i* model to a final software product by means of a precise model transformation process. Finally, we use a case study that is based on a photographic agency system in order to illustrate our approach.

Keywords: Goal-Oriented Requirements Engineering, i*, Requirements Integration, Object Oriented Method, Model-Driven Development.

1 Introduction

The success of computer applications increasingly depends on a good understanding of the system requirements. Currently, a requirements specification should include, in addition to software specifications, business models, domain models and other kinds of information that describe the context in which the intended system will operate. During early stages of requirements engineering, it is necessary to identify and specify how the intended system meets the organizational goals, why the system is needed, what alternatives were considered, what the implications of the alternatives are for the stakeholders, and how the stakeholders' interests and concerns might be addressed.

Hence, Goal-Oriented Requirements Engineering (GORE) stood out because it is mainly concerned with the stakeholders intentions and their rationales. Several works on GORE have being proposed: KAOS [6], i* framework [18], MAPS [15], Non-Functional Requirements (NFR) framework [5]. In all of them, requirements modeling appears to be a core process. However, how to go from requirements models to the corresponding software product is still an open question. To answer this question,

we advocate the use of GORE with Model-Driven Development (MDD) [17], two complementary model-based approaches.

Thus, we need a requirements model with such a structure that facilitates the specification of model transformations for the automatic generation of conceptual models used in MDD approaches. In this context, since present-day technologies (such as ATL or QVT) propose the specification of model transformations driven by metamodels, the use of GORE approaches is a suitable alternative, given that they have an abstract syntax formalized by a metamodel specification [3][11]. Among these GORE approaches, we selected the *i framework [18] because it is a consolidated modeling technique [8] with good tools support [10].

In this paper we propose guidelines to generate, from an i* requirements model, a conceptual model that is used as input of a MDD process for software products generation. This MDD process is based on the OO-Method approach [14]. We have chosen OO-Method as a reference MDD technology because it allows the complete generation of the final application from a conceptual model, and it has been successfully applied to industrial software development by means of the OlivaNova tools [4].

Therefore, this work proposes the generation of an OO-Method conceptual model from an i* requirements model based on a set of transformation guidelines, aiming to improve the quality of the models used on the development of information systems, and consequently to obtain better software products. To illustrate these guidelines, we have selected a real problem that was solved in the context of the PROS Research Center [12]: a Photography Agency. The main contribution of our work is to present an approach that provides a solution for filling the gap between GORE proposals and MDD proposals. The approach presented in this paper is part of a wider effort, which investigates the use of MDD techniques to define a full software process that covers the long path that goes from requirements modeling to the corresponding final software product.

This paper is organized as follows: Section 2 briefly describes the background considered in our proposal. Section 3 outlines the transformation process and a set of guidelines to perform it. Section 4 describes some relevant related works. Finally, Section 5 summarizes our work and points out open issues.

2 Background

This section starts with the presentation of an illustrative case study used as example across the paper to clarify the involved concepts. Later, the main features of the participant technologies (i* and OO-Method) are presented.

2.1 The Case Study

The photography agency is dedicated to the management of photo reports and their distribution to publishing houses. This agency operates with freelance photographers, which must present a request to the production department of the photography agency. This request contains: the photographer personal information, a description about the owned equipment, a brief curriculum, and a book with the performed photographic reports. An accepted photographer is classified in one of three possible levels for

which minimum photography equipment is required. For this, the technical department creates a new record for the photographer, and saves it in the photographer's file. For each photo report presented by a photographer a new record with a sequential code is created. This record has the price that the publishing houses must pay to the agency, which is established according to the number of photos and level of the photographer. Furthermore, this record has a descriptive annotation about the content of the report. The commercial department establishes according to the level of photographers, the price that will be paid to the photographers and the price that will be charged to the publishing house for each photo.

2.2 The i* Goal-Oriented Requirements Framework Overview

The goal-oriented modeling has proved to be an efficient means of capturing the '*Whys'* and establishing a close relationship with the '*Whats*' [10][16]. GORE is concerned with the use of goals for eliciting, elaborating, structuring, specifying, analyzing, negotiating, documenting, and modifying requirements.

The i* framework [18] captures the intentional requirements using strategic relationships among actors. The term actor is used to generically refer to any unit for which intentional dependencies can be ascribed. Actors are intentional, in a sense that they do not simply carry out activities and produce entities, but also they have desires and needs. Actors are also strategic, since they are not merely focused on meet their immediate goals, but also they are concerned about longer-term implications of their structural relationships with other actors, for instance, opportunities and vulnerabilities.

The i* framework offers two congruous models: the *Strategic Dependency* (SD) model and *Strategic Rationale* (SR) model. The SD model is focused on external relationships among actors. It includes a set of nodes and connecting links, where nodes represent actors (*depender* and *dependee*) and each link indicates a dependency (*dependum*) between two actors. In the SD model, the internal goals, knowhow, and resources of an actor are not explicitly modeled. In this model, we distinguish among four types of dependency links, based on the type of *dependum*: *goal*, *resource*, *task*, and *softgoal* dependencies. A *goal* in the i* context is a condition or state of concerns that the actor would like to obtain. A *resource* is a physical or informational entity that must be available for an actor. A *task* specifies a particular way of doing something, which can be decomposed in small sub-tasks. Finally, a *softgoal* is a condition that the actor would like to achieve, but some criteria are not well-defined. In general, the *softgoal* is associated to non-functional requirements.

The SR model (such as the example i* model presented in Fig. 1) expands the description of a given actor and all rationales involved on its intentions, providing support for modeling the reasoning of each actor about its intentional relationships. In addition to the dependencies present in the SD model, three new type of relationships are incorporated in the SR model: (i) *task-decomposition links*, which describe what should be done to perform a certain task (e.g. *To process a work request* task); (ii) *means-end links*, which suggest that a task (e.g. *To process a work request* task) is a means to achieve a goal (*A photographer´s work request be processed* goal); (iii) *contributions links*, which suggest how a model element can contribute to satisfy a *softgoal*. In particular, in our example, we do not have this last link. With the SR

model (Fig. 1), we capture some of the rationales involved in a photographer´s work request approval. For instance, a photographer must present a work request to the Production Department in order to have a work opportunity. In Fig. 1, this is represented by the resource dependency link between the *Photographer* actor and the *Production Dep* actor. To achieve this goal, the photographer must compose a work request that contains: a description of his/her equipment, a brief curriculum, and a book with his/her photographic reports. Finally, this request is processed by the *Production Dep.* actor.

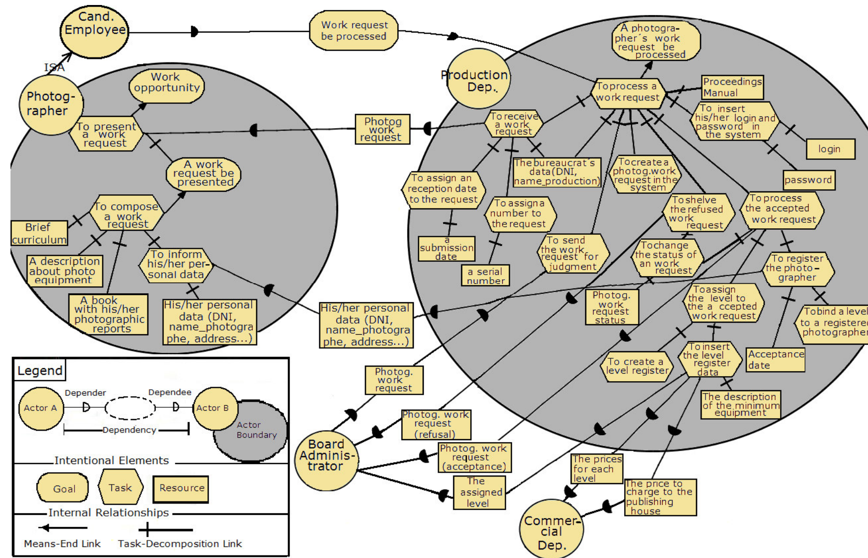

Fig. 1. The SR model of the Photographer work request

Despite an empirical evaluation has indicated that there are some problems with the i* framework [7], this framework is considered to be efficient enough to deal with complex actors, their organizational environment, and all rationales involved in their relationships. It allows the clear and simple statement of actors, their goals and the dependencies among them. Therefore, with the i* intentional views, we can obtain a rich model. However, the problem still remains: from the requirements model, how can we obtain the corresponding software product? For this, we propose the use of models transformations to integrate i* and Model-Driven Development approaches.

2.3 The OO-Method Model-Driven Development Approach Overview

Models help to understand a complex problem and its potential solutions through abstraction [17]. Thus, MDD methods have been created to take advantage of models in development processes, by using concepts that are much less bound to the underlying implementation technology and are much closer to the problem domain. This makes it easier to specify, understand, and maintain software systems. Besides, with MDD methods it is possible to achieve the automatic generation of the final products

by means of models transformations. Among different MDD approaches, we have selected the OO-Method approach as the reference MDD approach for our proposal.

The OO-Method MDD approach separates the application and business logic from the platform technology, allowing the automatic code generation from the conceptual representation of the software systems [14]. The OO-Method production process (Fig.2) is comprised of four models: the *Requirements Model*, the *Conceptual Model*, the *Execution Model*, and the *Implementation Model*.

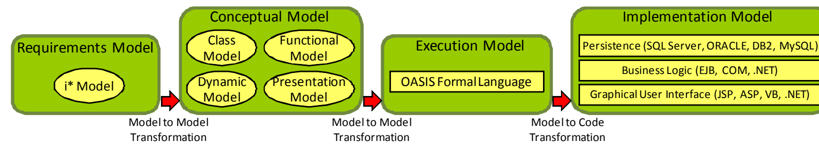

Fig. 2. The OO-Method Software Production Process with i*

In our proposal, we consider to use the i* framework as the OO-Method Requirement Model in order to capture the organizational context and the actors intentions. Next, from the defined i* model, an initial OO-Method Conceptual Model is inferred, which is used for the generation of the final software product. At this point, it is important to mention that the main modeling constructs provided by OO-Method Conceptual Model are the same as UML provides [17]. This situation also occurs in several object-oriented MDD approaches. Therefore, the results presented in this paper can be generalized to other MDD approaches based on the use of UML-like models.

The OO-Method Conceptual Model captures the static and dynamic properties of the functional requirements of the system in a *Class Model*, a *Dynamic Model*, and a *Functional Model*. The conceptual model also allows the specification of the user interfaces in an abstract way through the *Presentation Model*. These four models represent the different views of the whole conceptual model that has all the details needed for the generation of the corresponding software application. The complete definition of the elements of the OO-Method Conceptual Model is described in detail in [14]. From the models that comprise the OO-Method conceptual model, the class model is the most important, and the other models are defined (or derived) from this central model. Fig. 3 shows the original class model of the case study presented in section 2.1. In this model, the classes with their respective attributes and relationships, including all the necessary details, are introduced.

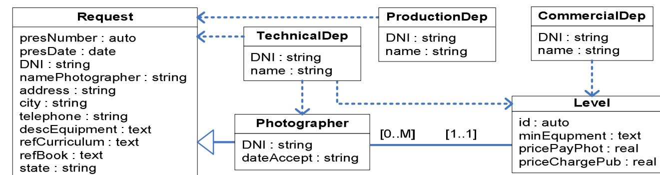

Fig. 3. The conceptual class model of the Photographic Agency System

In the OO-Method Conceptual Model, certain classes can access properties and invoke services provided by other classes (or by the same class). The permissions that a class has over other classes are defined by *agent relationships* (see dashed lines in

Fig. 3). In OO-Method, the associations are binary, i.e., they only have one or two participant classes (one class in the case of recursive associations).

With the OO-Method Execution Model, it is possible to perform the transition from the problem space (represented by the conceptual model) to the solution space (the corresponding software product).

Finally, the Implementation Model fixes the mappings between the conceptual constructs and their corresponding software representations in a target implementation platform, for instance C# or Java. The OO-Method approach has been successfully applied to the software industry with a MDD tool created by the enterprise CARE-Technologies [4]: *OlivaNova The Programming Machine*.

3 From i* Requirements Models to Conceptual Models

We propose a transformation process to make it possible the transformation of the i* models into a preliminary conceptual model for the OO-Method approach, presented in Fig. 4 with the Business Process Modeling Notation (BPMN). For lack of space, we only use the i* SR model.

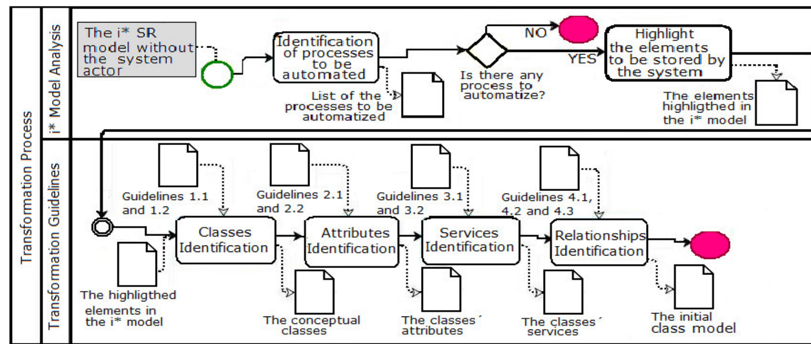

Fig. 4. The transformation process modeled with BPMN

Initially, we analyze the goals defined in the SR model (see Fig.1) to capture the organizational processes that we want to automate. Then, we highlight the intentional elements that are related to these processes (goals and tasks in the i* model). Those elements will be related with the information and/or entities to be stored by the intended system. From the list of identified intentional elements we obtain an initial conceptual model through model transformation rules, based on nine guidelines.

3.1 The i* Model´s Analysis

According to the transformation process (Fig. 4), this phase is comprised by the following activities: (i) identification of processes to be automated in the intended information system from the i* SR model; and (ii) highlighting of the essential issues that must be stored at the intended system.

Identification of the process to be automated. In this activity, we deal with the goals in the i* SR model. We seek for processes (tasks in a means-end link) that operationalize the intended goal, making it reachable. Therefore, in our case study, we recognize the following goals: *Work opportunity* for the *Photographer* actor; and, *A photographer´s work request be processed* by the *Production Dep*. actor. There are processes as means to reach those ends, respectively: *To present a work request*; and *To process a work request*. From these, we decided to automate the last process,

Highlighting the essential elements. For each process to be automated, we analyze the respective task-decomposition tree inside the actor boundary(e.g. the task – *To process a work request*). Through this analysis we highlight all essential elements that must be stored in the intended system, at the considered process (see Fig. 5). These selected elements are all those related with the process to be automated. Then, the selected elements from the i* model will be translated to elements of the Class Model using the transformation guidelines presented below.

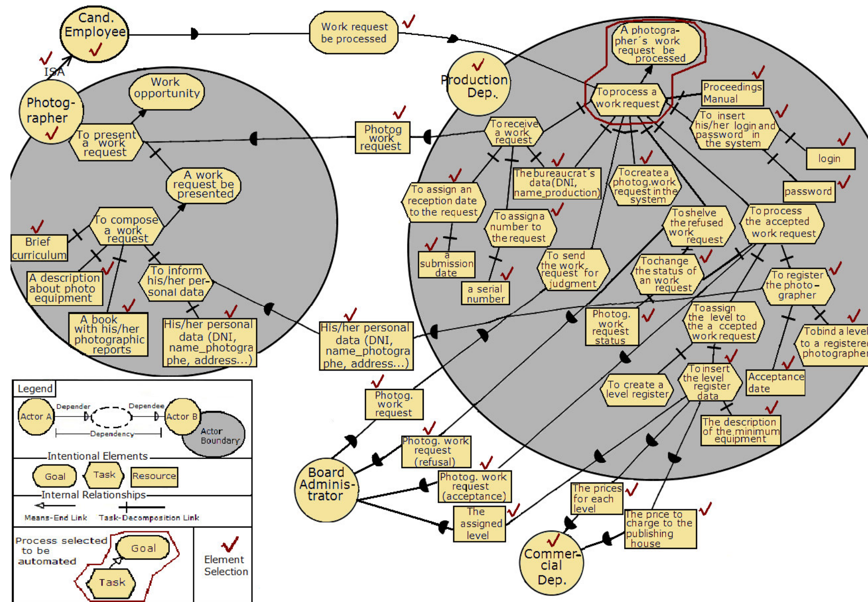

Fig. 5. The highlighted SR model of the Photographer work request

3.2 The Transformation Guidelines

In this phase, the guidelines to construct the OO-Method Conceptual Model from the i* model are presented. These guidelines are grouped in four activities: (i) the class identification; (ii) the attributes identification; (iii) the services identification; and (iv) the relationships identification. We have selected the class model as target because it is the core model of the OO-Method Conceptual Model.

Identification of classes**.** This activity deals with the identification of the main classes that should be in the class model. Indeed, in this step we are looking for the actors and the resource elements at the i* models. We do this because, by definition [18], an actor is an active entity that carries out actions to achieve goals by using its capabilities, while a resource is an entity (physical or informational), a finished product of some deliberation-action process. Therefore, both are related with the class concept.

Guideline 1.1: Related to actors of the i* model

We have two options to make the transformation from the i* actors to the class model.

(i) Looking for the *actors* whose data must be captured and maintained at the intended system. In this case, the actor is transformed into a *class* in the class model. For instance, in Fig. 5 we found *Cand. Employee* (Candidate Employee), *Photographer, Production Dep.* and *Commercial Dep.* actors, which are transformed in classes (see Fig. 6).

(ii) Actors that do not satisfy the previous statements are not transformed into elements of the class model. For instance, there is no need to save any information of the *Board Administrator* actor (see Fig. 5).

Guideline 1.2: Related to resources of the i* model

In relation to the resources elements, and considering the dependencies between actors, we propose the following transformation:

(i) Resources representing a physical entity that must be maintained in the system. In this case, the resource is transformed into a class. For instance, in our case study we have the following resources: *Photog. work request*, *Photog. work request (refusal)*, *Photog. work request (Acceptance)*, *Proceedings Manual* and *Assigned level*. These elements are transformed in the classes *WorkReqPhoto*, *ReqRefused*, *ReqAccepted*, *ProcManual* and *Level*, respectively (see Fig. 6).

The identification of attributes of classes. For each class obtained by the transformation of an i* actor or resource, their attributes must be identified. To do this, the main branch related to the process to be automated is analyzed. Usually, this branch corresponds to the means task that satisfies the intended goal (means-end link at i* models). Therefore, the resources that represent an informational entity will be our main target because they represent the attributes of the related class. These resources are transformed into attributes of the previously generated classes. To do this, we analyze the actor boundary. For instance, in the case study (see Fig. 5) we select the class *ProductionDep* using the Guideline 1.1 (item i) and the class *WorkReqPhoto* (Guideline 1.2). To ask for a job (means task for the goal *Work opportunity*) the photographer must compose the corresponding work request. Thus, in order to define the attributes of a class obtained by a resource mapping, we must also look for the task related with this resource.

Guideline 2.1: Related to classes generated from actors of the i* model

The following elements must be analyzed to obtain the attributes of the classes generated from actors of the i* model:

(i) A resource (informational entity) inside of the transformed actor. If this resource expresses information about the actor, then it is transformed into an attribute of the class. For instance, in Fig. 5, the actor *Photographer* (transformed into the class *Photographer*) must inform his/her personal data. Therefore, the resource

personal data will be transformed into attributes of the class *Photographer*. Finally, the details of *personal data* are the attributes of the target class: *DNI*, *name photographer*, *address*, *city*, *telephone*, and *brief curriculum*.

(ii) A resource outside the transformed actor (a resource dependency where the mapped actor is the *dependee* actor). This resource is transformed into an attribute of the mapped dependee actor. For instance, the dependency resource *personal data (DNI, name_photographer, address, etc.)* that will be available by the actor *Photographer* (the *dependee* actor in this dependency).

Guideline 2.2: Related to classes generated from resources of the i* model
For each resource that was transformed into a class, it must be considered if the resource is an internal element (it is inside of the actor boundary) or if the resource is related to a resource dependency link.

(i) If the resource is inside of an actor boundary (see in Fig. 5), then the attributes are inferred (according to the analyst experience) from the task that produces this resource or a sub-task (of another task) that produces informational entities related to the state of the analyzed class. In our case study (Fig. 5) we have the resource *Proceedings Manual* as an example for this case.

(ii) If the resource is a *dependum* element (it is outside of the actor boundary, in a resource dependency link), then both sides of this dependency must be analyzed to capture any informational entity (attribute) about the involved resource. This will be done by analyzing the tasks inside of the graphs of the *depender* and *dependee* actors. The task that produces the resource (inside of the *dependee* actor) and the tasks that need the resource (inside of the *depender* actor). For instance, in the case study, the class *WorkReqPhoto* (Fig. 6) is related with a resource dependency between the actors *Photographer* and *Production Dep*. (see Fig. 5). From the side of the *dependee* actor (*Photographer*), the graph with the task *To present a work request* as root is analyzed. A deep search is performed to find resources (leafs of the searched graph) related with the analyzed resource. From this search, we find the resources *Brief curriculum*, *a description about photo equipment* and *a book with his/her photographic reports* which will be transformed into attributes of the class *WorkReqPhoto* (the dependum element at Fig. 5). From the side of the *depender* actor (*Production Dep*.), we do the same. Thus, by analyzing the task *To receive a work request*, the resources *submission date* and a *serial number* are found. These resources are also transformed into attributes of the class *WorkReqPhoto*. However, the task *To receive a work request* is a sub-task of another task. Therefore, we must rise a level in our quest, and make the deep search in other branches of the graph. By the analysis of the task *To change the status of a work request* the resource *Photog. work request status* is found. This resource is also transformed into an attribute of the class *WorkReqPhoto*.

Fig. 6 shows all the attributes obtained after applying these transformation guidelines on all the classes that were derived from resources of the i* model (Fig. 5).

The identification of services of classes. At this point, the tasks of the i* SR model and their possible decompositions are inspected (deep search). In the i* framework, a task specify a particular way of doing something. When a task is described as a subcomponent of a (higher) task, in a hierarchy of tasks, this restricts the higher task to

that particular course of action (a task-decomposition link at the SR model). Moreover, from the practical experience, a task in the i* model generally is responsible for a goal's satisfaction and/or for the resource's production. We must remember that a service describes a specific behavior of the objects of a class, and, in the OO-Method approach, a service can be atomic (*Event*) or a composition of other services (*Transaction*). The events related to creation, deletion, and modification of class instances are always created by default in the Olivanova tool. Thus, to identify the other services of a class, we propose the following guidelines:

Guideline 3.1: Identification of services of a class generated from an actor
The internal sub-graphs must be analyzed, which generally are a routine responsible for the satisfaction of a goal of the corresponding actor. From these sub-graphs, only must be considered the tasks that must be stored at the intended information system.

(i) If the task represents a change in the state of an object that occurs instantly, then this task is transformed into an event of the generated class. In Fig. 5, we do not find this situation because atomic services are not represented according to the considered abstraction level in the i* model.

(ii) If the task represents a service that groups other services, then this task is transformed into a transaction of the generated class.

Guideline 3.2: Identification of services of a class generated from a resource
In this case, we are looking for tasks that are used or produced by the transformed resource, and identifying if the resource is inside or outside an actor.

(i) If the task represents an instantaneous change in the state of an object, then this task is transformed into an event of the generated class. For instance, for the resource *Photog. work request*, inside of the *depender* actor (see *Production Dep.* in Fig. 5), there is a task called *To create a work request in the system*. This task is transformed into an event of the class *WorkReqPhoto.* In addition, the generated event allows the generation of new instances of the class. On the other side, at the *dependee* actor (*Photographer*), the task *To present a work request* is also transformed into an event of the class *WorkReqPhoto*.

(ii) If the task represents a service that groups other services, then this task is transformed into a transaction of the generated class. For instance, in the Photographic Agency example (see Fig. 1), an accepted work request must be processed according to the task *To process the accepted work request.* This task is decomposed on three sub-tasks: *To register the photographer*, *To assign the level to the accepted work request*, and *To change the status of a work request*. Therefore, a new transaction must be created in the class *WorkReqPhoto* to represent the execution of these three tasks.

These two guidelines must be applied to all classes generated from the i* model.

The identification of relationships between classes. In this point, the three basics relationships of object-oriented approaches are considered: generalization / specialization, association, and aggregation. However, it is important to remark that i* mainly focuses on representing strategic concerns by means of intentional elements and their relationships. Therefore, the information of each relationship of the i* model must be analyzed to derivate the kind of relationships among the generated classes.

Guideline 4.1: Identification of Generalization/Specialization relationships among generated classes

We must considerer two possibilities:

(i) If the class is derived from an actor and there is an inheritance relationship between actors of the i* model (the *is-a* relationship), then this relationship is automatically transformed into a generalization in the class model. For instance (see Fig.5), we found the *is-a* relationship between the actors *Candidate Employee* and *Photographer*. This relationship is represented as a generalization between the corresponding generated classes of the class model (Fig. 6).

(ii) If the class is derived from a resource and the inheritance relationship is not explicit at the i* models, then we must analyze the processes (tasks) involved in the production of this resource. For instance, from the resource dependencies between the actors *Production Dep.* and *Board Administrator*, we can observe that a work request may be accepted or refused. These work requests were transformed into the classes *WorkReqPhoto, ReqRefused* and *ReqAccepted*. Since *ReqRefused* (refused photographer's work request) and *ReqAccepted* (accepted photographer's work request are a *WorkReqPhoto*, then we generate an inheritance relationship between these classes (see Fig. 6).

Guideline 4.2: Identification of Association relationships among generated classes

We must considerer the following possibilities:

(i) For two classes generated from i* actors, if there is any dependency link between the two transformed actors, then an association between the corresponding classes is automatically generated in the class diagram. For instance, the actor *ProductionDep* was transformed into a *class* and it must also be associated with the service of other class. The *Photographers* present their request to the production department (class *ProductionDep*). Therefore, an association is generated between these two classes (Fig. 6).

(ii) If there exists a resource dependency link where the *dependum*, the *depender* and *dependee* actors were transformed into classes, then associations are automatically generated between these classes. However, if there is any generalization relationship between one of these classes (resulting from the actors transformations), then the association is defined with the corresponding father class. For instance, for the class *Photographer* (from the actor *Photographer*) it must be defined an association to the class *WorkReqPhot* (from the *dependum* resource *Photog work request*). However, since there is a generalization between the classes *Photographer* and *CandidateEmp*, then the involved association is defined between *CandidateEmp* and *WorkReqPhoto* (Fig. 6). The same occurs for the association defined between the classes *WorkReqPhoto* and *ProductionDept*.

(iii) For a resource dependency link where the *dependum* is transformed into a class attribute and the *depender* and *dependee* actors are transformed into classes, an association is generated among the classes generated from actors and the class that has the attribute generated from the involved resource. For instance, in the Fig. 5 there is a resource dependency link (*The prices for each level* resource) between the actors *Production Dep.* and *Commercial Dep*. The respective resource was transformed into an attribute of the class *Level*. Thus, an association is generated between the classes *CommercialDep* and *Level*, and between the classes *Level* and *ProductionDep* (Fig. 6).

(iv) For a class resulting from the transformation of an internal resource, an association is created between this resource class and the class resulting from the transformation of the respective actor boundary (the one that contains the resource). For instance in Fig. 5, inside the actor *Production Dep.* there is a resource (*Proceedings Manual*) that was transformed into a class. Therefore, an association is generated between the respective classes into the class diagram (Fig. 6).

Guideline 4.3: Aggregation relationship between generated classes
We must considerer two possibilities:

(i) If the class is generated from an actor and there is an aggregation relationship between actors of the i* model (the *is-part-of* relationship), then this relationship is automatically transformed into an aggregation in the class model.

(ii) If the class is generated from a resource and the aggregation relationship is not explicit at the i* models, then the internal behavior of the actor that is directly associated with a resource that was transformed into a class must be analyzed.

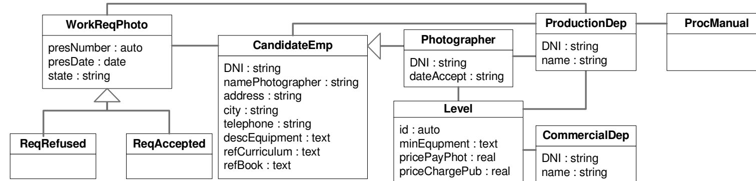

Fig. 6. The class model obtained from the application of the proposed guidelines

3.3 Discussion

In this paper, we have presented nine guidelines that are used to go from i* requirement models to the class model of a MDD approach. These guidelines were systematically designed in accordance with the i* framework [10]. To illustrate the application of the guidelines, we have manually applied the guidelines to a Photography Agency case study. Even though the guidelines have been designed in the OO-Method MDD context, many conceptual constructs of the OO-Method class model are similar to the constructs of the other object-oriented methods. For this reason, the proposed guidelines can be generalized to allow the application to other MDD methods.

With respect to the automation of the guidelines, we classified the guidelines as *automatic* (it is not necessary any intervention of the analyst), *semi-automatic* (some decisions of the analyst are required), and *manual* (they application completely depends of the analyst expertise). Thus, the guidelines *1.1*, *4.1i*, *4.2*, and *4.3i* are automatic, the guidelines *1.2*, *2.1*, *2.2*, *3.1* and *3.2* are semi-automatics, and finally, the guidelines *4.1ii* and *4.3.ii* are manuals.

Analyzing the Photography Agency case study, we can state, through a comparison between the class model originally constructed for the case study (Fig. 3) and the class model generated by the application of the proposed guidelines (Fig. 6), that some elements were incorrectly represented in the original class model (Fig. 3). For example, a *photographer* was considered as a *request*, which is incorrect because these are different objects. Furthermore, the class *TechnicalDep* (Fig. 3) was merged

with the class *ProductionDep* because during the specification of the i* model (Fig.1) we identify that both roles have common tasks from the organizational viewpoint, so that we decide to merge all the tasks in the actor *Production Dep*. (*ProductionDep* in the generated class model – Fig. 6). A new generalization/specialization relationship with two new classes (*ReqRefused* and *ReqAccepted* classes) was created for the class *WorkReqPhoto*. While in the original class model, there was only one attribute in the class *WorkReqPhoto* to indicate whether the proposal was rejected or accepted. With the representation obtained in the generated class model, it is possible to define specific attributes related to accepted and refused requests, which is not possible in the original class model. Hence, we may conclude that GORE approaches, as the i* framework, are very rich in terms of intentions and their rationales, which must be reflected in later development stages. Thus, taking GORE approaches as starting point of the software development process, and using MDD techniques to reach the final software product, an improved solution for software development is obtained.

It is important to note that in our proposal the quality of the GORE models directly affects the generation of correct conceptual models of the MDD approach. This quality mainly depends of the experience of the analyst in the problem domain and in the usage of the modeling technique. The abstraction level is also dependant of the viewpoints and the focus of the analyst. In this proposal, we assume that the i* models are correct, complete, and that do not present defects (omissions, inconsistency, erroneous facts, ambiguous, etc.). Therefore, applying the proposed guidelines, we can infer the basis of the conceptual model without introducing any modeling defect. We know that this assumption is unrealistic. For this reason, we are also working in proposals to evaluate the quality of the i* models in order to improve the application of our proposal in MDD environments.

Despite of the positive and important aspect of our work that concludes that it is possible to incorporate goal concepts in a MDD approach we also highlighted some other points which are being investigated: (i) the i* framework is more expressive at a high abstraction level than the OO-Method conceptual model, consequently, the guidelines only consider a subset of the i* framework; (ii) some important concepts for the OO-Method Conceptual Model are not captured by the i* models, since the abstraction levels are different into these approaches (for instance, additional relationships information such as roles or cardinality), therefore, additional information is required to correctly infer the corresponding OO-Method concepts; (iii) since certain guidelines are not automatic, the transformation cannot be fully automated; (iv) the traceability between requirements and conceptual models is not considered in the transformations; (v) the guidelines only provide a partial generation of the class model and additional formalization is required for a correct software product generation.

4 Related Works

Some strategies based on i* have been proposed with the aim of reducing the gap between requirements phase and the software development phase.

The proposal presented in [13] is a methodological approach that enables the generation of conceptual and requirements models from organizational descriptions. To do this, two strategies were considered: (1) to extend the organizational model with

monitoring plans and concerned objects, and (2) to define guidelines to establishing correspondences among business requirements and the conceptual model of the system. This proposal uses the particular version of i* defined in *Tropos* [9] and defines a set of complex steps to obtain a partial conceptual model definition. By contrast, we use the i* version disposable at *i* Guide* [10] to define guidelines for the direct inference of OO-Method conceptual constructs from i* models. This provides a more straightforward way for the generation of an initial conceptual model, which facilitates the application of our proposal to MDD processes.

In previous works ([1][2][16]) we have proposed a process to derive late requirements specifications specified in *pUML* (precise Unified Modeling Language) from early requirements model represented in *i** framework. In [1], we proposed a set of guidelines to go from i* models to class diagrams in order to obtain the conceptual model of the business model, which differs from the approach presented in this paper, that generates the conceptual model of the information system. In [2] and [16] we intended to generate scenarios and use cases represented with UML from i* models. To do this, a set of guidelines that helps the requirements engineer to determine the existence of potential use cases from the business model specification were proposed. However, the use case generation is not our goal in this paper, since we intend to directly transform an i* model into a conceptual model of the OO-Method MDD approach.

5 Conclusions and Future Works

In this work we consider the combination of a specific GORE approach (i*) and a specific MDD approach (OO-Method) to go from the requirements models to the corresponding software product. Both GORE and MDD are based in the use of models, and we believe that they can complement each other. This proposal is part of a wider work that is related to the use of MDD techniques to define a full software process that covers the long path that goes from goal-oriented requirements modeling to a final high-quality software product.

In this paper, we presented a set of transformation guidelines that are applied to the industrial MDD approach OO-Method, in order to facilitate the transformation of an initial i* intentional requirement model into an automatically generated software product. In addition, since OO-Method is an object-oriented MDD approach many of the concepts analyzed can be reused by other object-oriented MDD approaches.

The automatic generation of the complete final software products is performed by means of a precise model transformation process. Therefore, as future work, we plan to apply the guidelines to other case studies in order to evaluate the correctness and completeness of our proposal. In addition, we plan to formalize the guidelines using metamodeling techniques and models transformations technologies in order to be automatically applied and to preserve the requirements traceability through the models. Finally, we also consider the definition of extensions for the i* framework in order to facilitate the capture of new features.

Acknowledgments. This work has been developed with the support of CNPq and CAPES research grants, BIT initiative, and MEC under the project SESAMO TIN2007-62894.

References

1. Alencar F.: Mapping an Organizational Model in Precise Specification. Ph.D. Dissertation, Department of Informatics from University of Pernambuco. Recife, Brazil (1999)
2. Alencar F., Pedroza F., Castro, J., Amorim, R.: New Mechanism for the Integration of Organizational Requirements and Object Oriented Modeling. In Proc. of the VI Workshop on Requirements Engineering (WER 2003), Piracicaba, Brazil. pp.109-123 (2003)
3. Ayala C., Cares C., Carvallo J.P, Grau G., Haya M., Salazar G., Franch X., Mayol E., Quer C.: A Comparative Analysis of i*-Based Agent-Oriented Modeling Languages. In Proceedings of the 17th SEKE, pp. 657-663 (2006).
4. Care Technologies Company: OlivaNova Suite. Available at www.care-t.com. Last access: Jul. (2009)
5. Chung, L., Nixon, B., Yu, E. and Mylopoulos, J.: Non-Functional Requirements in Software Engineering. Kluwer Academic Publishers (2000)
6. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-Directed Requirements Acquisition. Science of Computer Programming 20(3) (1993)
7. Estrada, H., Rebollar, A. M., Pastor, O., Mylopoulos, J.: An Empirical Evaluation of the i* Framework in a Model-Based Software Generation Environment. In CAiSE’06. LNCS 4001, Springer-Verlag , pp. 513-527 (2006)
8. Grau, G., Franch, X., Ávila, S.: J-PRiM: A Java Tool for a Process Reengineering i* Methodology. RE 2006: p.352-353 (2006)
9. Giorgini P., Mylopoulos J.,Sebastiani R.: Goal-Oriented Requirements Analysis and Reasoning in the Tropos Methodology. In Engineering Applications of Artificial Intelligence, Elsevier 18(2), March (2005)
10. Abdulhadi, S.:i* Guide v.3, Aug. 2007 Available at: http://istar.rwth-aachen.de/tiki-view_articles.php. Last access: Jul. (2009)
11. Lucena, M. ; Santos, E. ; Silva, M. J. ; Silva, C. ; Alencar, F. ; Castro, J. F. B. :Towards a Unified Metamodel for i*. In: 2nd IEEE Int. Conference on Research Challenges in Information Science (RCIS'08), Marrakech. Proceedings of the RCIS'08, pp. 237-246 (2008).
12. Marín, B., Giagchetti, G., Pastor, O.: The Photography Agency: A case study of the OO-Method Approach. Technical Report DSIC-II/13/08, Universidad Politécnica de Valencia, Valencia, Spain (2008)
13. Martínez, A.: Conceptual Schemas Generation from Organizational Models in an Automatic Software Production Process, PhD Thesis, Universidad Politécnica de Valencia, Valencia, Spain (2008)
14. Pastor, O. and Molina, J. C. Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling, Springer-Verlag 1st ed., Springer, New York, New York (2007)
15. Rolland C., Prakash N., Benjamen A. A multi-model view of process modeling. Requirements Engineering, 4 (4), pp. 169-187 (1999)
16. Santander, V., Castro, J.: Deriving Use Cases from Organizational Modeling. 10th Anniversary IEEE Joint International Conference on Requirements Engineering (RE 2002), Essen, Germany. September, pp. 32-42 (2002)
17. SELIC, B. The Pragmatics of Model-Driven Development. IEEE Software, 20, pp. 19–25 (2003).
18. Yu, E.: Modelling Strategic Relationships for Process Reengineering, PhD Thesis, University of Toronto, Toronto, Canada (1995).
19. BPMI.org: Business Process Modeling Notation; OMG Available Specification. Object Management Group, version 1.1. Available at http://www.bpmn.org/. Last access Sep 2009 (2008).