A Model Transformation Approach to Derive Architectural Models from Goal-Oriented Requirements Models

Marcia Lucena[12], Jaelson Castro[1], Carla Silva[3], Fernanda Alencar[1], Emanuel Santos[1]

[1]Federal University of Pernambuco (UFPE) – Recife/PE – Brazil
[2]Federal University of Rio Grande do Norte (UFRN) – Natal/RN – Brazil
[3]Federal University of Paraíba (UFPB) – Rio Tinto/PB – Brazil

{mjnrl, jbc,ebs}@cin.ufpe.br, fmra@ufpe.br, ctaciana@ccae.ufpb.br

Abstract. Requirements engineering and architectural design are key activities for successful development of software systems. Both activities are strongly intertwined and interrelated, but many steps toward generating architecture models from requirements models are driven by intuition and architectural knowledge. Thus, systematic approaches that integrate requirements engineering and architectural design activities are needed. This paper presents an approach based on model transformations to generate architectural models from requirements models. The source and target languages are respectively the i* modeling language and Acme architectural description language (ADL). A real web-based recommendation system is used as case study to illustrate our approach.

Keywords: Requirements engineering, Architectural design, Models Transformation

1 Introduction

The Requirements Engineering (RE) [12] and Software Architecture (SA) [8] are initial activities of software systems development that have been emerging both in research as in practice. Currently, software systems present characteristics such as increased size, complexity, diversity and longevity. Thus, their development must consider proper requirements elicitation and modeling approaches, as well as use systematic architectural design methods. A great challenge is the development of systematic methods for building architectural design that satisfies the requirements specification. Some efforts have been made to understand the interaction between RE and SA activities [3], [1]. Therefore, many approaches claim that there is a semantic gap between these two activities. However, with the widely use of iterative and incremental software development process as the *de facto* standard, a strong integration between requirements and architectural design activities can facilitate traceability and the propagations of changes between these models efficiently [15]. In this context, we can highlight the twin peaks model [14], which emphasizes the co-development of requirements and architectures, incrementally elaborating details.

Recognizing the close relation between architectural design and requirements specification [5], the Model-driven Development (MDD) [11] appears as an effective way to generate architectural models from requirements models by using model transformations rules, in which the correlation between requirements and architectural models can be specified accurately. Thus, in this paper, we show an approach to generate architectural models from requirements model that includes horizontal and vertical transformations rules. The horizontal transformations are applied to requirements models and results in other requirements models closer to architectural model. While the vertical transformations map these resulting requirement models in architectural models. The main contribution is on the vertical transformation rules that complement the horizontal transformation rules presented in [13]. In our approach, architectural models are described using Acme ADL [6], which provides a simple structural framework for representing architectures, whereas requirements models are described using the modeling language offered by the i* [18], a goal-oriented approach to describe both the system and its environment in terms of strategic actors and social dependencies among them.

This paper is organized as follows. Section 2 introduces our case study and overviews the main concepts of the i* and Acme languages. Section 3 presents our approach based on model transformation rules. Section 4 describes related works. Finally, Section 5 summarizes our work and points out open issues.

2 Background

This section presents our case study and briefly reviews the requirements modeling and architectural description languages used in our approach.

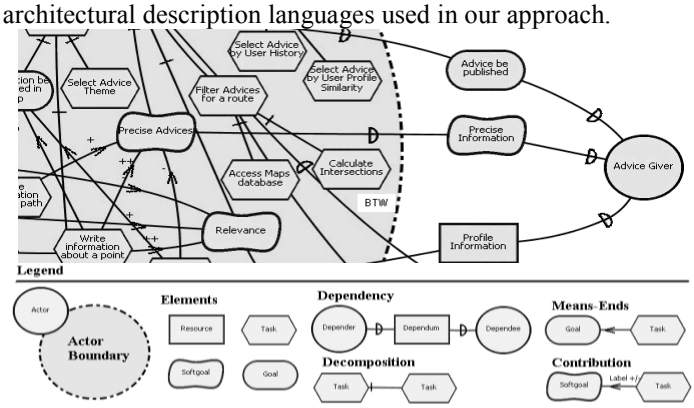


Fig. 1 Partial BTW Strategic Rationale Model

2.1 BTW Project

The BTW-UFPE project [2], presented in the SCORE contest held at ICSE 2009 [16], is used to illustrate our approach. BTW consists in a route-planning system that helps users through the recommendation of advices about a specific route searched by the user. This information is posted by other users and might be

filtered to provide the user only with relevant information about the place that he/she intends to visit.

The BTW-UFPE team generated artifacts that include i* requirement models. We chose this project by two reasons: it is a real case study that resulted in a software system; and the produced i* models are not large but have enough complexity to illustrate the benefits of our approach. Fig. 1 shows a partial SR model for BTW. Its complete models can be found in [2].

2.2 The Source: i* Requirements Goal Model

i* defines models to describe both the system and its environment in terms of intentional dependencies among strategic actors [18].There are two different models: the Strategic Dependency (SD) describes information about dependencies and the Strategic Rationale (SR) describes details about each actor.

The SR model complements the information provided by the SD model by adding internal details for each strategic actor to describe how the dependencies are accomplished. In i* models, a depending actor is called a depender, and an actor that is depended upon is a dependee. Fig. 1 presents dependencies between Advice Giver and BTW actors. Considering the *Advice be Published* goal dependency, the BTW actor is the depender actor whereas the Advice Giver actor is the dependee actor of this dependency. BTW represents the software system to be developed. Thus, an actor can depend upon another one to achieve a goal, execute a task, provide a resource or satisfy a softgoal. Softgoals are associated to non-functional requirements, while goals, tasks and resources are associated to system functionalities [19]. Actor's internal details also include tasks, goals, resources and softgoals, which are further refined using task-decomposition, means-end and contribution links. The *task-decomposition* links describe what should be done to perform a certain task (e.g., the relationship between the *Filter Advices for a route* task and the *Access Maps database* task). The *means-end* links suggest that one intentional element can be offered as a means to achieve another intentional element (e.g., relationship between the Select Advice by User History task and the Relevant Advice be chose goal). Finally, the contributions links suggest how a task can contribute (positively or negatively) to satisfy a softgoal (e.g., the relationship between the *Write information about a point* task and the *Precise Advices* softgoal).

In this paper, we are concerned with how to manage the internal complexity of the software actor (BTW) and how to produce architectural models from it.

2.3 The target: ACME architecture models

A set of elements are important when describing instances of architectural designs. According to [17], these elements include Components, Connectors, Interfaces, Configurations and Rationale. Acme ADL [6] supports each of these concepts but also adds ports, roles, properties, and representations. Besides, Acme has a textual and a graphical language. Acme Components represent computational units of a system. Connectors represent and mediate interactions between components. Ports correspond to external interfaces of components. Roles represent external interfaces of connectors. Ports and roles (interface) are points

of interaction, respectively, between components and connectors. Systems (Configurations) are collections of components, connectors and a description of the topology of the components and connectors. Systems are captured via graphs whose nodes represent components and connector and whose edges represent their interconnectivity. Properties are annotations that save additional information about elements (components, connectors, ports, roles, and systems). Representations allow a component, connector, port, and role to describe its design in detail by specifying a sub-architecture that refines the parent element. Properties and representations could be associated to the rationale of the architecture, i.e., information that explains why particular architectural decisions were made, and for what purpose various elements serve [17].

Architectural design is not a trivial task, even using specific concepts to describe architecture. It depends on the expertise of the architects and on how they understand the requirements. To make this task more systematic, we propose a MDD approach to derivate an early architectural design from requirements models.

3 Architectural Design by using Model Transformations

To generate Acme architectural models from i* models, we propose a process composed of three major activities: (i) analysis of internal elements, (ii) application of horizontal rules and (iii) application of vertical rules. This process recognizes that i* models are intrinsically complex and this complexity need to be managed. The first two activities are concerned to this while the last activity is concerned with the development of architectural design. To perform these activities, it is required to use, respectively: (i) conditions to guide the software actor's decomposition, (ii) model transformation rules to generate modular i* models, and (iii) model transformation rules to generate Acme architectural models. This process is semi-automatic, since interventions of the requirements engineer are likely to be necessary to take some decisions. The activity (i), in particular, uses some conditions to assist the requirements engineer to choose elements that can be moved to another software actor to balance the responsibilities of an actor. The other activities can be developed with few interventions.

In this paper, we concentrate on the activity (iii), as the activities (i) and (ii) have already been presented in [13]. We only present them briefly in the Section 3.1 and 3.2, respectively.

3.1 Analysis of Internal Elements

The decomposition criterion is based on the separation and modularization of elements that are not strongly related to the application domain. For example, in the BTW SR model (Fig 1), which captures the web recommendation system requirements [2], we can identify those elements that are not fully related to the application domain (recommendation). At this point, a requirements engineer must perform the analysis. Some sub-graphs internal to the BTW actor are considered independent from the recommendation application domain and, therefore, can be moved to new software actors. Thus, sorting out the independent elements into other actors can improve reusability and maintainability of system

specification at the requirements level. In fact, considering the BTW SR model, the following elements could be used as part of a system of a different application domain: Map to be Handled, User Access and Information to be published.

3.2 Application of Transformation Rules

In this activity, an appropriate horizontal transformation rule must be applied. The rule to be applied depends on the type of relationship between the elements to be moved and the elements that will remain in the original system actor. The general purpose of the horizontal rules is to delegate internal elements from the system actor to other actors [13]. This delegation establishes a dependence relationship between the new actors and the original actor, maintaining the semantics of the original model in the resulting model. These horizontal rules will be briefly presented in this section (for more details see [13]).

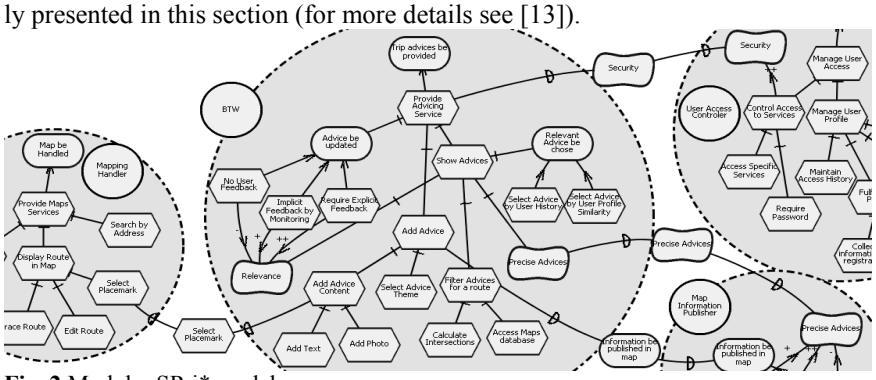


Fig. 2 Modular SR i* model

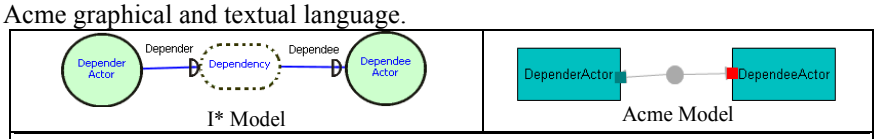
HTR1 is a transformation rule that moves a sub-element present in a task-decomposition to another actor. HTR2 considers the situation where the sub-graph to be moved has the root element as a "means" in a means-end relationship. After applying rules HTR1 and HTR2, the resulting model may not be in conformity with the i* notation. In this case, we need to use a corrective rule, such as HTR3. This rule was defined to preserve the information about contribution links and maintain the information about contribution links and coherence of i* models as it is proposed in [9]. And HTR4 is applied when the sub-graph to be moved out has a sub-element shared with other sub-graphs.

After applying the horizontal rules to the selected elements in Analysis activity, three new actors are created related to sub-graphs of *Map be Handled*, *User Access be Controlled* and *Information be published in Map* goals (Fig. 2). As these new actors receive the name of their elements (goal or task), we suggest to use a specific noun related to the domain of these elements. Thus, we have respectively *Mapping Handler*, *User Access Controller*, and *Map Information Publisher*. In this activity, the main rules used were HTR3 and HTR4.

As the horizontal rules are applied, the i* model is transformed into an i* model closer to an early architectural design. This modularized i* model is an entry to the activity of vertical rules application. All new created actors, the main software actor and their dependencies among each other will be used in the vertical transformation rules. Next section presents the rationale behind the vertical transformation rules to produce Acme architectural models from i* requirements models.

3.3 Generating architectural models

The models associated with different activities of the software development process are created using specific model description language. Therefore, how models will be generated from a stage to another depend on how the transformation rules are defined considering the main elements of each involved modeling language. We start to establish the vertical transformation rules considering only actors and dependencies to map i* elements to Acme elements, as it is presented in [4]. Fig. 3 shows a generic mapping without considering the type of dependency between actors and dependencies, in i*, to components and connectors in Acme graphical and textual language.




```
1 System ClientServer = {
2    Component DependerActor  = {
3        Port port1  = {
4            Property Required : boolean = true;        }  }
5    Component DependeeActor  = {
6        Port port2  = {
7            Property Provided : boolean = true;        }   }
8    Connector ConnDependency  = {
9        Role depender  = {    }
10       Role dependee  = {    }    }
11     Attachment DependerActor.port1 to connDependency.depender;
12    Attachment DependeeActor.port2 to ConnDependency.dependee;}
```

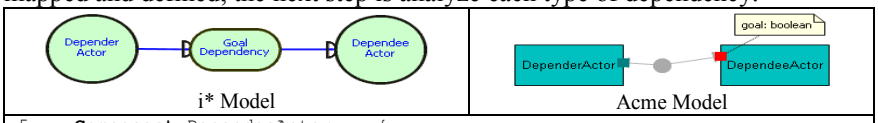
Fig. 3 Mapping a generic dependency between i* actors to ACME

A component in software architecture is a unit of computation or a data store having a set of interaction points (ports) to interact with external world [17]. An actor in i* is an active entity that carries out actions to achieve goals by exercising its knowhow [18]. The actor representing the software establishes a correspondence with modules or components [7]. In addition, an actor may have as many interactions points as needed. Hence, an actor in i* can be represented in terms of a component in Acme (Fig. 3).

Connectors are architectural building blocks that regulate interactions among components [17]. In Acme, connectors mediate the communication and coordination activities among components. In i*, a dependency describes an agreement between two actors playing the roles of depender and dependee, respectively [4]. Thus, we can represent a dependency as an Acme connector. Interfaces are points of access among components and connectors. However there are not ports in i*, but points where dependencies interact with actors. Depending on role of dependency (depender or dependee) we can know when an actor is a depender or a depended upon actor. Hence, the roles of depender and dependee are mapped to connector roles that are comprised by the connector (Fig. 3). Thus, we can distinguish between required ports (where the actor is a depender) and provided ports (where actor is a dependee). For instance, Fig. 3 shows the use of property Required (line 4) and property Provided (line 7) indicating the direction of communication between the DependerActor and DependeeActor components. Therefore, in i* a depender actor depends on a dependee actor to accomplish a type of dependency. In Acme, a component needs that another component carries out a service and the requisition of this service is done by a required port, while the result of this service is done by a provided port, thus, a connector allows the

communication between these ports. A component offers services to another component using provided ports and a component require services using its required port.

Applying this mapping in BTW project (Fig. 2) we will have four components: *BTW, Mapping Handler, User Access Controler,* and *Information map Publisher*. Each dependency is mapped to a connector and the roles of their connectors will be depender and dependee according the direction of dependency. For instance, when an actor has at least one dependency as a dependee, its equivalent component will have at least one provided port (Fig. 3). Therefore, the *Mapping Handler* component will have a provided port considering the *Placemark* resource dependency. Having all components, ports, connectors and roles mapped and defined, the next step is analyze each type of dependency.



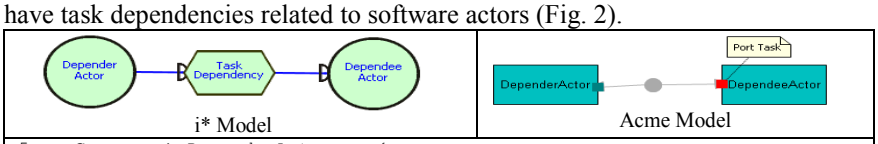

```
5    Component DependeeActor  = {
6        Port port2  = {
7            Property Provided : boolean = true;
8            Property goal: boolean;        }    }
```

Fig. 4 Mapping a goal dependency to ACME

In i*, the type of dependency between two actors describes the nature of the agreement established between these actors. There are four types of dependency: goals, softgoals, tasks and resources. Each type of dependency will define different architectural elements in connectors and in ports that play their interfaces. A goal dependency is mapped to a Boolean propriety related to a provided port of the component that offers this port (Fig. 4, line 8). This property represents a goal that this component is responsible to fulfill by using a provided port.

The type of property is Boolean in order to represent the goal satisfaction (true) or no satisfaction (false). Applying this goal dependency mapping in BTW case study implies that BTW and Information Publisher components will add new Boolean properties in respectively provided ports.

A task dependency represent that an actor depends on another to execute a task [18] and that a task describes or involves processing [7]. Since port in Acme port correspond to external interfaces of components and offer services, as it was said before. Hence, a task dependency is mapped directly to a provided port of component that offers this port (Fig. 5, line 6). In our BTW example we do not have task dependencies related to software actors (Fig. 2).

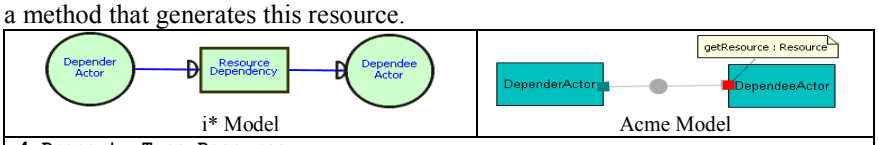



```
5    Component DependeeActor  = {
6        Port Task  = {
7            Property Provided : boolean = true;    }    }
```

Fig. 5 Mapping a task dependency to Acme

In a resource dependency, an actor depends on another actor to provide information. Therefore, a resource dependency is mapped to a return type of a property of a provided port (Fig. 6, line 8). This return type represents the type of the resulting product that an operation related to some service that the component is responsible to perform. This mapping is to show that while a task is generate by an actor in a component, it is generated by an element inside of port. In Fig. 2,

there is one case of resource dependency that is placemark resource dependency. Thus, the provided port of *Mapping Handler* component receives a property with a method that generates this resource.

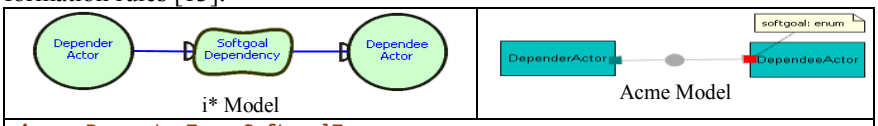



```
4  Property Type Resource;
5     Component DependeeActor  = {
6         Port port2  = {
7             Property Provided : boolean = true;
8             Property getResource : Resource;    }    }
```

Fig. 6 Mapping a resource dependency to Acme

A softgoal dependency is similar to goal dependency but its fulfillment cannot be defined precisely. A softgoal is related to a non-functional requirement that will be treated by a task or a softgoal more specific. Hence, a softgoal dependency is mapped to a property with enumerated type present into the port that plays the dependee role of the connector (Fig. 7, line 9). This enumerated type is used to describe the degree of satisfaction of the softgoal. For the BTW example, the *Publisher* component has a provided port that interface with Precise Advices connector by the dependee role. This provided port will have a property softgoal defined as enumeration type. The same mapping is done with the security dependency between *BTW* and *Access User Controller* component.

Acme graphical language uses labels to highlight added elements used in textual part. However, using AcmeStudio tool, information of types of dependency only is presented in textual language. Each mapping presented will be formalized by vertical transformation rules following the same structure of horizontal transformation rules [13].




```
4     Property Type SoftgoalType =
5          enum {make,somePos,help,unkown,break,someNeg,hurt};
6     Component DependeeActor  = {
7         Port port2  = {
8             Property Provided : boolean = true;
9          Property softgoal: SoftgoalType;    }    }
```

Fig. 7 Mapping a softgoal dependency to Acme

4 Related Work

We highlight two goal-oriented approaches [20][21] and a MDD approach [15]. The SIRA approach [20] focuses on a systematic way to assist the transition from requirements to architecture. It describes a software system from the perspective of an organization in the context of the Tropos methodology. The requirements and r architecture models are described in i*. An organizational architectural style is chosen based on the non-functional requirements. Thus, an architectural model is created considering similarities of elements of requirements and organizational style. In our proposal, we also use i* goal model as input model, but we modularize these models to reach an architectural configura-

tion. Moreover, we also present a systematic way to treat non-functional requirements and we use a target model based on a generic architectural language (Acme). In [20] it is not clear how is structured in the architectural model.

Lamsweerde [21] defines a method to generate architectural models from KAOS requirements models. In this approach, specifications are gradually refined to meet specific architectural constraints of the domain and an abstract architectural draft is generated from functional specifications. The resulting architecture is recursively refined to meet the various non-functional goals analyzed during the requirements activities. It is used KAOS models, which consist of a graphical tree and a formal language. In our approach, we use another goal model as input model, the i* models. The relation between i* notation and software architecture facilitate the mapping between these models, while this is not occur in KAOS models. In contrast, that approach provides guidelines to refine an initial architecture applying architectural styles and patterns, while in our approach we provide a preliminary architecture.

In [15] is proposed a set of mapping rules between the AspectualOV-graph (AOV-graph) and the AspectualACME, an architecture description language. Each element (goal/softgoal/task) present in an AOV-graph is mapped to an element of AspectualACME, depending on the position that each element is in the graph hierarchy. The information about the source of each element is registered in the properties of a component or a port. These properties make it possible to keep traceability and change propagation between AspectualACME to AOV-graph models and vice-versa. We also propose a set of mapping rules between a goals model i*, but considers the non-aspectual version of ACME ADL.

5 Conclusions and Future Works

Our approach generates an initial architectural model described in Acme, from i* requirements models. To achieve this, it was necessary to balance the responsibilities of a system actor, delegating them to other new system actors. A set of horizontal rules proposed in [13] were used to generate modular i*models which are closer to architectural design. From the modular i* model we derive an Acme model through a set of mappings between the concepts of both languages. These mappings were based on [4] which the purpose was to map i* architectural models to architectural models described using UML-RT, since i* was not conceived to be an ADL. Besides, UML-RT is a language specific to Object Orientation paradigm. The difference from this work to ours is that we are concerned in creating an architectural design from a requirement specification and not just mapping between languages for architectural description. Since our mapping relates requirements and architectural models, allowing better traceability and propagation change. Furthermore, using a more general architectural language led us to propose more generic mapping rules, which in turn can serve as a guide to derive architectural models in other ADLs. We assessed our approach using a web recommendation system (BTW) that is a real system developed for a Software Engineering contest at ICSE 2009 [2].

An issue that needs to be further explored is the systemic nature of some NFRs. Currently we are also defining formally our transformation rules (horizontal and vertical) using Alloy language to ensure that the resulting models will be

well formed. The use of Alloy will enable us to define the transformation rules and verify their soundness. However, in order to implement our proposal we rely on a specific model transformation language, namely ATL.

Other future work includes automating this approach through the implementation of our transformation rules in the Istar Tool [10], a tool based on MDD and Eclipse Platform. This will allow us to investigate the scalability of our approach in some real life complex projects.

References

1. Berry, D. M. Kazman, R., Wieringa, R.: 2nd Intl Ws on From SofTware Requirements to Architectures (STRAW'03) at ICSE'03, Portland, USA, (2003)
2. Borba, C, Pimentel, J., Xavier, L.: BTW: if you go, my advice to you Project. https://jaqueira.cin.ufpe.br/jhcp/docs/. Jul (2009)
3. Castro, J., Kramer, J.: 1st Intl. Workshop on From SofTware Requirements to Architectures (STRAW'01) at ICSE'01, Toronto, Canada (2001)
4. Castro, J., Silva, C., Mylopoulos, J.: Modeling Organizational Architectural Styles in UML. CAiSE 2003: 111-126, (2003)
5. Deboer, R., Vanvliet, H.: On the similarity between requirements and architecture, Journal of Systems and Software (2008)
6. Garlan, D., Monroe, R. Wile, D.: ACME: An Architecture Description Interchange Language". Proc. of the CASCON 97, (1997)
7. Grau, G., Franch, X.: On the Adequacy of i* Models for Representing and Analyzing Software Architectures. ER Workshops 2007: 296-305. (2007)
8. Hofmeister, C., Nord, R., Soni, D.: Applied Software Architecture. Addison-Wesley. (2000)
9. Horkoff, J.: Using i* modeling for evaluation, Master's Thesis, University of Toronto, Department of Computer Science (2007)
10. IStarTool Project: A Model Driven Tool for Modeling i* models. Available at http://portal.cin.ufpe.br/ler/Projects/IstarTool.aspx, Jul (2009)
11. Kleppe, A., Warmer, J., Bast, W.: MDA Explained - The Model Driven Architecture: Practice and Promise. Addison-Wesley. (2003)
12. Kotonya, G., Sommerville, I.: Requirements Engineering: Processes and Techniques. Wiley, John & Sons Inc. (1998)
13. Lucena, M., Silva, C., Santos, E., Alencar, F., Castro, J.: Applying Transformation Rules to Improve i* Models. In: SEKE 09, pp 43-48 USA (2009)
14. Nuseibeh, B.: Weaving Together Requirements and Architectures. IEEE Computer 34(3), 115-117 (2001)
15. Silva, L., Batista, T., Garcia, A., Medeiros, A., Minora, L.: On the Symbiosis of Aspect-Oriented Requirements and Architectural Descriptions. LNCS Vol. 4765(2007)
16. The SCORE 2009. Available at http://score.elet.polimi.it/index.html. Jul (2009)
17. Taylor, R. N., Medvidovi, N., Dashofy, I. E.: Software Architecture: Foundations, Theory, and Practice. John Wiley & Sons, (2009)
18. Yu, E.: Modeling Strategic Relationships for Process Reengineering. Ph.D. thesis. Department of Computer Science, University of Toronto, Canada (1995)
19. Yu, E. et al.: i-star Tutorial in RE'08, IEEE Computer Society, pp 1-60, Spain (2008)
20. Bastos, L., Castro, J.: From requirements to multi-agent architecture using organisational concepts. ACM SIGSOFT Software Engineering Notes 30(4): 1-7 (2005)
21. Van Lamsweerde, A.: From System Goals to Software Architecture, LNCS, Vol 2804/2003, p. 25-43 (2003)