In *Proceedings of the First International Workshop on Requirements, Intentions and Goals in Conceptual Modelling* (RIGiM'07). Pages: 296-305.

On the Adequacy of *i** Models for Representing and Analyzing Software Architectures

Gemma Grau, Xavier Franch

Universitat Politècnica de Catalunya
c/ Jordi Girona 1-3, Barcelona E-08034, Spain.
{ggrau, franch}@lsi.upc.edu

Abstract. In order to work at the software architecture level, specification languages and analysis techniques are needed. There exist many proposals that serve that purpose, but few of them address architecture and requirements altogether, leaving a gap between both disciplines. Goal-oriented approaches are suitable for bridging this gap because they allow representing architecture-related concepts (components, nodes, files, etc.) and more abstract concepts (goals, non-functional requirements, etc.) by using the same constructs. In this paper we explore the suitability of the *i** goal-oriented approach for representing software architectures. For doing so, we check its properties against the ones suitable for Architecture Description Languages and we define some criteria for solving the unfulfilled aspects in representing the architectures. This paper assumes basic notions on *i**.

1. Introduction

Many researchers have realized that, to obtain the benefits of an architectural focus, software architecture must be provided with its won body of specification languages and analysis techniques [17]. The purpose of Architecture Description Languages (hereafter, ADLs) is to demonstrate the properties of the system at its early stages and minimize the cost of errors. For doing so, ADLs have to provide adequate abstractions and, at the same time, enough level of detail for establishing the properties of interest. However, despite the need for and benefits of specifying non-functional properties, there is a notable lack of support for them in existing ADLs [18].

It is possible to observe that most requirements frameworks provide mechanisms for model analysis in order to inform further decisions but, despite the similarities in both fields, there is a recognized gap between requirements and architectures. According to [12] this gap is mainly due to the different representation of concepts in requirements and in architectures. This leads to the use of new techniques and models for bridging the gap between requirements and architectures. From this point of view, goals are an adequate formalism for representing the concepts on both disciplines with the required level of detail. Because of that, there are many approaches that advocate the use of goal-oriented models [16], and in particular, the *i** notation [20], for representing software architectures (see [13]).

In *Proceedings of the First International Workshop on Requirements, Intentions and Goals in Conceptual Modelling* (RIGiM'07). Pages: 296-305.

Following this tendency, in [9], [10], we presented SAR*i*M, a Software Architecture Reengineering *i** Method, that aims at assessing the selection of a system architecture. SAR*i*M is an evolution of PR*i*M [8] a method for exploring and evaluating process and system alternatives by representing its requirements with *i**. Consequently, SAR*i*M uses the *i** constructs to model the current architecture and, then, supports the exploration of alternative architectures by means of different *i** models, which are evaluated using structural metrics [7]. SAR*i*M copes with the architectures exploration and evaluating process using the *i** constructs for representing architectures and, so, works at the requirements level. For a reliable application of the method, we need to make an adequate use of *i** when modelling the architectures. However, although most of the previously mentioned work uses *i** for modelling software architectures, no consensus has been reached for representing architectural concepts within *i**. As far as we know, only [2] and [5] propose some correspondence between *i** and the agent architectural description language, but they do not address how the use of *i** fulfils the desirable properties for being an ADL.

In order to ensure that *i** is adequate for representing software architecture, we have checked its properties against the desiderata for software architectures presented in [19]. As a result, we have found that *i** accomplishes the principles:

- **Composition.** *i** allows describing a system as a composition of independent components and connections, where components are represented by *i** actors and connections by *i** dependencies.
- **Abstraction.** *i** makes possible to describe the software system at different levels of detail (for instance, early requirements, late requirements, architectural design or detailed design [3]).Thus, the components and their interaction can be described with or without software architecture in a way that clearly prescribes their abstracts roles in a system.
- **Analysis.** There are several proposals on how to analyse *i** models [13], being the most commonly used goal-reasoning techniques [20] and structural metrics [6], [7].

Thus, it is possible to perform rich and varied analyses of architectural descriptions. However, there still some open aspects that remain open:

- **Reusability.** As it is highlighted in [1], the inherent freedom of the *i** language makes most of the groups working with *i** to use their own criteria and constructs. Thus, despite *i** could make possible to reuse components, connectors, and architectural patterns in different architectural descriptions, the different variations of *i** used and the lack of consensus in representing architectural concepts, could make this reuse difficult. On the other hand, most of the *i** models are build from the scratch and, although some work uses *i** architectural patterns [2], [3], [14] there are no catalogues and directives for reusing them.
- **Configuration and Heterogeneity.** For the same reasons stated above, there is often a lack of prescriptiveness when constructing the models and there exist different works that propose different uses of the constructs, which damages configuration and heterogeneity.

In order to address these issues we have analysed the *i** constructs against the most relevant modelling features proposed in [18] for the classification and comparison of ADLs. In those aspects not fulfilled by *i**, we have used our experience in *i** modelling and architecture modelling, and other work related on *i** [13] to provide a

suitable solution. As a result, in section 2, we present a proposal for representing components and connectors by means of actors and dependencies. Then, in section 3 we extend this proposal for representing software architecture configurations and we show how they can be refined through different kinds of models allowing reuse. Finally, in section 4, we present the conclusions and future work.

2. Adapting the *i** Constructs to Software Architectures

In order to adapt the *i** constructs to the needs for representing software architectures, we propose to use the building blocks of the architecture description established in the ADL classification and comparison framework proposed in [18]. The main modelling features are components, connectors, and architectural configurations. For components and connectors, the main elements to take into account are their interface, types, semantics, constraints, evolution, and non-functional properties (see [18] and further sections for a description of the elements). In the following sections we propose a way to adapt the *i** constructs for satisfying these modelling features.

2.1. Using Actors for Modelling Components

A Component in a software architecture is a unit of computation or a data store [18]. In ADLs, components have an interface which contains a set of interaction points (also called *ports*) that allows interacting with the external world. In *i**, actors are the most intuitive way to represent components, being their ports the points where the dependencies are connected to the actor. As we explain in the next section, the different types of dependencies have different architectural meaning and, accordingly, we only consider as *ports* those links related with resource and task dependencies. Following this criteria, an *i** actor may have as many *ports* as needed and, as dependencies are bidirectional, we can distinguish between input *ports* (where the actor is the *depender*) and output *ports* (where the actor is the *dependee*).

The functionality of the components is encapsulated into reusable blocks, which are abstracted by means of component types. Actors can be distinguished by a label with their name. As *i** actors can represent different kinds of entities (i.e., stakeholders, software systems, hardware sensors, etc.) we consider adequate to add an attribute to the actors for indicating their *structural type*. Our proposed initial set of structural types is software, hardware, human, and organization. More structural types can be added if needed, and also, it is possible to further divide each structural type into subtypes, eventually in more than one level. For instance, for software actors we may distinguish commercial off-the-shelf components (COTS), modules, layers, etc.

The semantics of the components define the component behaviour. Semantics are needed to perform analysis, enforce architectural constraints, and ensure consistent mappings. In the semantics of the *i** actors, we have different intentional types that may be used for these purposes by defining the abstraction level of the components:

- A *role* represents a service or group of related services that can be supplied by a certain component. For instance, in Fig. 1, we present the model of a COTS system

for a meeting scheduler, where we identify the roles of *Meeting Scheduler*, *Message Delivery*, and *Antivirus*.

- A *position* represents types of components available to cover one or more roles. For instance, in the COTS market we identify that for a Meeting Scheduler System there are packages whose primary purpose is to act as a *Meeting Scheduler* but also provide mailing facilities as a *Message Delivery*. Therefore, we can use a *Communicated Meeting Scheduler* position (see Fig. 1).
- An *agent* represents an specific component that can be integrated into the software architecture. For instance, if in the COTS market there is a specific component that accomplishes the functionality required by the *Communicated Meeting Scheduler* position, the agent (e.g. the ACME Meeting Net in Fig. 1) will occupy this position.

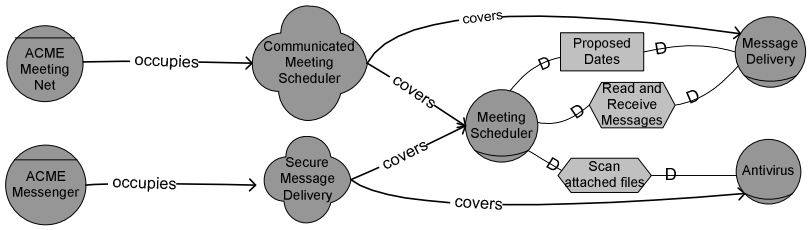

Fig. 1. Excerpt of *i** model for a Meeting Scheduler showing some roles, positions and agents

2.2. Using Dependencies to Represent Connectors

Connectors are architectural building blocks used to model interactions among components and rules that govern those interactions [18]. The interface of the connectors is the set of interactions points between the connector and the components attached to it. In *i** connectors are represented as dependencies where the direction of the dependency indicates the role of each connector, being one the *depender* and the other the *dependee*.

Components communication, coordination, and mediation decisions are encapsulated by the connector types. The *i** dependencies are associated with a *dependum* with a label which indicates the name of the object or concept that is shared among the components. Dependencies have an already associated *structural type* (goal, task, resource and softgoal), which interpretation changes according to the expectative of the *depender* and the *dependee* upon the *dependum*. Thus, the *depender* depends on the *dependee* to bring about a certain state in the world (goal dependency), to attain a goal in a particular way (task dependency), for the availability of a physical or informational entity (resource dependency) or to meet some non-functional requirement (softgoal dependency).

The connector semantics define the high-level model of the connectors' behaviour. In order to enforce the link between requirements and architectures when deciding the kind of a certain dependency, we propose to distinguish two different types of relationships to be represented with the four kinds of *i** dependencies: *intentional*, i.e. what behaviour a component expects from other part of the system, and *architectural*,

i.e. how one component communicates with other part of the system. The examples provided correspond to the Meeting Scheduler System represented in Fig. 2(a).

- **Intentional relationships.** They are the ones that involve human or organizational actors as dependee and/or depender. They represent the intentional needs of the actors upon the system as follows:
  - A goal dependencies state functional requirement over the system, e.g. the *Initiator* depends on the *Meeting Scheduler* for *Meeting be scheduled*.
  - Resource dependencies state flow of concepts, and remarkable some type of knowledge, or a concept, relevant for the domain that does not physically exist, e.g. the *Meeting Proposal* concept in the context of the meeting scheduler.
  - Softgoal dependencies state high-level non-functional requirements, which may refer to quality of service, development objectives, or architectural constraints over the components, e.g. the *Antivirus* shall ensure *Easy Configuration* to the *Administrator* actor.

  We remark that we do not model task dependencies at the intentional level, because they enforce that the *depender* provides a prescriptive procedure, and this is not done at this level. The use of the *i** dependencies when modelling early requirements, late requirements and architectural design in TROPOS [3] enforces this decision as they only use goals, softgoals and resources.
- **Architectural relationships.** They are the ones that occur between components of the system. In order to adhere with the architectural concepts, the semantics of the architectural relationships are defined by adapting the six CBSP architectural dimensions proposed in [12] into the *i** framework:
  - Goal dependencies model the elements that that describe system-wide features or features pertinent to a large subset of the system's components or connectors. For instance, the *Meeting Scheduler* depends on the *Message Delivery* for *Messages Received and Read*.
  - Task dependencies model the elements that describe or involve processing components. For instance, in fig. 2 (c), the task dependency *Scan attached files* states a service invocation request to the *Antivirus*.
  - Resource dependencies model the elements that describe or involve data components. For instance, the resource dependency *Proposed Dates* states information interchange between the software actors *Meeting Scheduler* and *Message Delivery*.
  - Softgoal dependencies model the elements that describe or imply data or processing component properties, bus properties or system properties. For instance, the *Meeting Scheduler* depends on the *Message Delivery* for a *Reliable Message Delivery*.

2.3. Constraints, Evolution and Non-functional properties

Architectural components and connectors also have to represent constraints, evolution, and non-functional properties. Constraints state properties about the system and ensure interaction protocols, intra-connector dependency and enforce usage boundaries. On the other hand, non-functional properties represent requirements for the correct

behaviour of the components and connectors. In *i**, both constraints and non-functional properties are represented as softgoal dependencies between the different kinds of actors: human and organization actors constrain the properties of the connectors (software actors) with softgoal dependencies upon them, whilst softgoals dependencies among software actors constrain the properties of the connectors.

In order to allow more formal constraints, the *i** constructs will have to be complemented with more specific work. For instance, [16] allows defining architectural constraints among goals in a formal way. Regarding evolution, the *i** constructs do not provide a way to store the modification of the actor's and connectors properties. However, the work reported in [4] proposes to use actor's inheritance (the *is-a* construct) to support the extension, refinement and redefinition of intentional elements, which can be used to record evolution of the actors (components) and can be extended to record the evolution of their dependencies (connectors).

3. Representing Architectural Configurations in *i**

Architectural configurations are connected graphs of components and connectors that describe the architecture structure [18]. According to [18], characteristics features at the level of architectural configurations are understandability, compositionality, refinement and traceability, heterogeneity, scalability, evolution, dynamisms, constraints, and non-functional properties. Some of these characteristics are directly supported by *i**, whilst others require the adoption on extended work on the field.

The graphical representation of *i** certainly enhances the understandability of the modelled architecture, mainly because the different shapes used and the fact that components and connectors are identified with a textual label, makes it very intuitive. Despite one of the open issues on *i** is how to improve scalability when the models get very big, regarding architectural representations *i** makes possible to work with the components at different levels of detail, i.e. by decomposing a component represented in a high level of detail, into several components and its relationships at lowers levels of details. This use of the *i** models also improves compositionality.

Refinement and traceability deal with the representation of a consistent refinement of the architecture and the traceability of changes. In a similar manner, evolution deals with the how architectures change to reflect and enable evolution of a family of software systems. This features are not directly addressed in the *i** framework, however, we propose to use the semantics provided by the *i** actors to distinguish 3 different models depending on its type. Therefore we may model the architecture at three different levels:

- **Roles model.** Provides a description of the different roles that we may distinguish in the system architecture (i.e., services or groups of related services). Dependencies may be intentional or architectural. However, in the roles model there are no task dependencies between software actors because at this level we do not know the specific architecture of the system neither the protocols that would be used. Consequently, we use goal dependencies to state the services that a software actor requires from another software actor. For instance, in Fig. 2(a) the *Meeting*

Scheduler depends on the *Message Delivery* for the goal *Messages Received and Read*, which, in turn, depends on the *Antivirus* for *Attached files scanned*.

- **Position model.** Identifies the different positions that exist in the architecture (i.e. the types of components available that cover one or more of the roles represented in the roles model). Usually this model does not state new dependencies, but, hides dependencies among roles which are covered by the same position. For instance, in Fig. 2(b) the *Communicated Meeting Scheduler* covers the roles of the *Meeting Scheduler* and the *Message Delivery* and so, the dependencies between them remain hidden. We remark that several different instantiations of the roles are possible (e.g. The *Secure Message Delivery* position covers the *Meeting Scheduler* and the *Antivirus* in Fig. 1). This leads to the exploration and modelling of different architectural solutions.

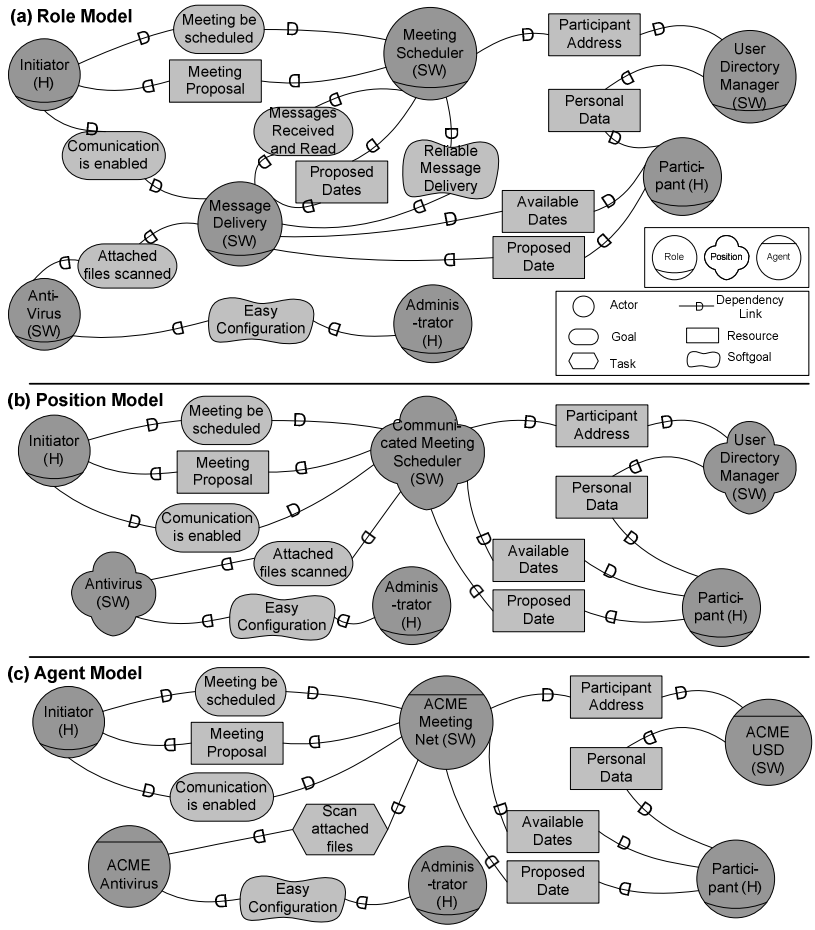

Fig. 2. Role model, position model, and agent model for an excerpt of the Meeting Scheduler

- **Agent model.** States the different agents that exist in the architecture (i.e. specific software component that can be integrated into the software architecture). In this model, dependencies denoting architectural relationships may refine the dependencies that exist on the role model, e.g., in Fig. 2(c) the *ACME Message Delivery* depends on the *ACME Antivirus* for the task *Scan attached files*, as we identify that the communication between those specific software components involves processing this data.

These three levels propose an agent-oriented perspective of the Software Architecture. However, as agent architectures do not have a direct correspondence to some types of software systems, the levels must be used as an abstraction mechanism. More precisely, these three levels provide a structure similar to the 4+1 view proposed in [15], in which the role model would represent the logical and process views, the position model the development view, and the agent model the physical view. A crucial point is that the mapping among models is one-to-many: a role model may have different position models bound, whilst a position model may have different agent models bound.

Another aspect mentioned in [18] for architectural configurations that it is not directly addressed by *i** is heterogeneity. This feature deals with how the language allows developing a software architecture by using pre-existing components and connectors of varying granularity possibly specified in different formal modelling languages, with varying operation system requirements, and supporting different communication protocols. In order to address this issue we propose to enrich the models by adding *attributes* to their modelling elements. Thus, attributes make possible to state characteristics of actors or dependencies. Attributes may be universal (e.g. *priority* in the case of the dependencies) or type-dependant (e.g., *programming language* in the case of software actors or *size* for resource dependencies). Attributes are useful not only to make *i** models more complete, but also for defining metrics and constraints among them.

In previous sections, we have seen that non-functional requirements can be bound to the components and connectors by means of softgoal dependencies that constrain their behaviour. However, at the configuration-level, non-functional requirements deal with the whole system and are used to ensure that the appropriate components and connectors are selected, perform analysis, enforce constraints, map architectural building blocks to processors, and aid in project management. In order to help the evaluation of the components and connectors and inform the analysis, we propose the use of structural metrics to evaluate properties and inform decisions. Structural metrics may depend on different parameters. Consider as an example data privacy. Each resource dependency in a position model identifies a danger, i.e. data interchange among different software components, so a first metric could be defined just as the total number of resource dependencies in the model:

$$DP(M) = \#k: k \in dependencies(M): type(k) = resource$$

This initial definition could be considered too simplistic. Refinements could be to consider the size of this resource:

$$DP(M) = \sum k: k \in dependencies(M) \wedge type(k) = resource: size(k)$$

The level of detail depends on how much effort the architect is able to put in the evaluation process and how confident is he/she in the result, for more information we refer to [6], [7].

Non-functional properties aimed to enforce constraints and global constraints over the system cannot be controlled with non-functional properties but, as with components and connectors, it is possible to use some of the extensions (for instance, the formal notation proposed in [16]) in order to represent them and check them.

4. Conclusions and Future Work

In this paper we analyse the *i** framework from an architectural point of view. As *i** allows a certain flexibility in the use of its elements, we have proposed some consensus in order to address architecture modelling. Our four main contributions are: 1) Clarifying the use of the *i** constructs for modelling components and connectors. As this is done by means of actors and dependencies we have provide an architecture-oriented semantics to them in order to help the process; 2) Adding the notion of role model, position model, and agent model in order to help traceability of the architectural representation; 3) Proposing the addition of attributes in the actors and dependencies of the models in order to store information for its analysis; and, 4) Suggesting the use of structural metric to analyse the properties of the final system.

Based on this study, we can conclude that it is possible to use *i**, a framework that includes a requirements language, for modelling architectures using the same constructs. The benefits of this are twofold. On the one hand, it is possible to represent and analyse a design architecture using the same constructs of their requirements model. In the other hand, architectures representation benefits from the notions of goals and softgoals, allowing a better representation of non-functional requirements.

We have to mention that, as shown in [19], it is very difficult to have an ADL that accomplish at its maximum extent all the mentioned features because it becomes too complex. Because of that we know in advance that *i** cannot be complete. Consequently, the modelling features needed and the techniques proposed have to be analysed in order to choose the most adequate. From this starting point, our future work will be directed to use this analysis as the basis for ensuring that the use of *i** in SAR*i*M is adequate for represent, explore and evaluate alternative architectures. We remark that, as we have proposed the addition of attributes on the *i** models (i.e. structural types for the actors, and attributes for all the elements), defining a model becomes laborious and time-consuming activity. However, the use of tool support facilitates this step and, because of that, we will adapt J-PR*i*M [11] to support them as we progress. The research agenda includes: 1) the creation of an ontology of the domain, putting together classical architectural concepts and goal-oriented ones; 2) the definition of a complete and widely applicable catalogue of metrics to be used in different experiences, eventually refining the framework outlined in [7]; 3) the definition and representation of usual architectural patterns in *i**; and, 4) the completion of a validation program, starting by retrospective analysis of existing cases and ending up with industrial case studies.

Acknowledgements. This work has been partially supported by the CICYT programme project TIN2004-07461-C02-01. Gemma Grau work is supported by an UPC research scholarship.

References

1. Ayala, C.P., Cares, C., Carvallo, J.P., Grau, G., Haya, M., Salazar, G., Franch, X., Mayol, E., Quer, C.: "A Comparative Analysis of *i**-Based Goal-Oriented Modelling Languages". In *Proceedings of SEKE 2005*. pp: 43-50.
2. Bastos, L.R.D., Castro, J.F.B.: "Enhancing Requirements to derive Multi-Agent Architectures". In *Proceedings of WER 2004*. pp. 127-139.
3. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: "TROPOS: An Agent-Oriented Software Development Methodology". *Journal of Autonomous Agents and Multi-Agent Systems*. May 2004. Volume 8, Issue 3.
4. Clotet, R., Franch, X., López, L., Marco, J., Seyff, N., Grunbacher, P.: "The Meaning of Inheritance in *i**". In *Proceedings of AOIS 2007*. pp: 651-666.
5. Faulkner, S., Kolp, M.: "Towards an Agent Architectural Description Language for Information Systems". In *Proceedings of ICEIS 2003*. pp. 59-66.
6. Franch, X.: "On the Quantitative Analysis of Agent-Oriented Models". In *Proceedings of CAiSE 2006*. LNCS 4001, pp. 495-509.
7. Franch, X., Grau, G., Quer, C.: "A Framework for the Definition of Metrics for Actor-Dependency Models". In *Proceeding of RE 2004*. pp. 348-349.
8. Grau, G., Franch, X., Maiden, N.A.M.: "A Goal Based Round-Trip Method for System Development". In *Proceedings of REFSQ 2005*. pp. 71-86.
9. Grau, G., Franch, X.: "ReeF: Defining a Customizable Reengineering Framework". In *Proceedings of CAiSE 2007*. LNCS 4495, pp. 485-500.
10. Grau, G., Franch, X.: "A Goal-Oriented Approach for the Generation and Evaluation of Alternative Architectures". To appear in *Proceedings of ECSA 2007*.
11. Grau, G., Franch, X., Ávila S.: "J-PR*i*M: A Java Tool for a Process Reengineering *i** Methodology". In *Proceedings of RE 2006*. pp. 352-353.
12. Grünbacher, P., Egyed, A., Medvidovic, N.: "Reconciling software requirements and architectures with intermediate models". *Software and Systems Modeling*, Vol. 3, Num. 3, August 2004. pp. 235-253.
13. The *i** wiki at: http://istar.rwth-aachen.de/. Last Accessed: June 2007.
14. Kolp, M., Giorgini, P., Mylopoulos, J.: "Organizational Patterns for Early Requirements Analysis". In *Proceedings of CAiSE 2003*. LNCS 2681. pp. 617-632.
15. Kruchten, P.B.: "The 4-1 View Model of Architecture". *IEEE Software*. November 1995, 12(6), p. 42-50.
16. van Lamsweerde, A.: "From System Goals to Software Architecture". In *Proceedings of SFM 2003*. LNCS 2804, pp. 25-43.
17. Medvidovic, N., Rosenblum, D.S.: Domains of Concern in Software Architectures and Architecture Description Languages. *In Proceedings DSL 1997*. pp. 199-212.
18. Medvidovic, N., Taylor, R.N.: "A Classification and Comparison Framework for Software Architecture Description Languages." *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70-93 (January 2000).
19. Shaw, M., Garlan, D.: *Software Architecture: Perspectives on an Emerging Discipline*. ISBN: 0-13-182957-2. Prentice Hall, 1996.
20. Yu, E.: *Modelling Strategic Relationships for Process Reengineering*. PhD. thesis, University of Toronto, 1995.