STREAM: A Strategy for Transition between Requirements Models and Architectural Models

Marcia Lucena 1, Jaelson Castro 2
1 DIMAP, Universidade Federal do Rio Grande do Norte (UFRN) Natal, RN - Brazil
2 CIn, Universidade Federal de Pernambuco (UFPE) Recife, PE - Brazil
marciaj@dimap.ufrn.br, jbc@cin.ufpe.br

Carla Silva 3
3 DCE/CCAE, Universidade Federal da Paraíba (UFPB) Rio Tinto, PB - Brazil
carla@dce.ufpb.br

Fernanda Alencar 4, Emanuel Santos 2
4 DES, Universidade Federal de Pernambuco (UFPE) Recife, PE - Brazil
fernanda.ralencar@ufpe.br, ebs@cin.ufpe.br

ABSTRACT
Requirements engineering and architectural design are strongly intertwined and interrelated software development activities. This paper presents a systematic approach to integrate requirements engineering and architectural design activities 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. Non-functional requirements (NFRs) are used to select among architectural solutions and architectural patterns. An e-commerce system is used as running example to illustrate our approach.

Categories and Subject Descriptors
D.2.1 [Software Engineering]: Requirements/Specifications – languages, methodologies.

General Terms
Measurement, Documentation, Design, Experimentation, Human Factors, Standardization, Languages, Theory.

Keywords
Relating requirements to architecture, Modeling of requirements and goals

1. INTRODUCTION
Many researchers associate the requirements with the problem-space and the architecture with the solution-space [1]. In fact, there is a semantic gap between requirements specifications and architectural design, which consists of conceptual differences between what to do (requirements) in opposition to how to do (architecture, design and coding). Despite these differences, the techniques and methods used for software development should include a systematic way for dealing with the relationship between requirement and architectural models.

In our work, requirements are defined in terms of i* models [2], a popular goal-oriented approach. The solution is outlined as an Acme architectural model [3]. Acme is a general-purpose Architectural Description Language (ADL), which can act as an interchange language.

This work proposes the STREAM (Strategy for Transition between Requirements models and Architectural Models). In order to illustrate the feasibility of our proposal, we have used the e-commerce system introduced in [4].

This paper is organized as follows. Section 2 overviews the main concepts of the i* and Acme languages, introducing our example. 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 reviews the i* requirements modeling language and the ADL used in our approach through a running example.

2.1 The source: i* requirements models
In order to illustrate the techniques used in this work, we review the Media Shop example [4]. Media Shop is a store that sells and ships different kinds of media items. To increase market share, Media Shop has decided to use the Medi@ system, a business to customer retail sales front-end on the Internet.

Strategic Rationale (SR) model [2] is used to expand the description of a given actor. In our example, the Medi@ actor, i.e. the software system to be developed, is expanded in Figure 1.

SR models include task-decomposition links that describe what are required to perform a certain task (e.g., the relationship between the Internet Shop Managed task and their sub-elements). SR models also include means-end links, representing intentional elements (usually tasks) offered as means to achieve another intentional element (usually goals). NFRs (also known as softgoals) analysis can be used to assist the choice among alternative means. For example, consider the relationship between the Use Standard Form, Use Form with Encryption tasks and the Identification Details Collected goal (Figure 1). Contribution links suggest that a task can contribute positively (help) or negatively (hurt) towards the achievement of a softgoal. For instance, in order to fulfill the Security softgoal, the best option is to choose Use Form with Encryption task, as it contributes positively (help) to its satisfaction (Figure 1). Having described the requirements, the challenge is to relate them to appropriate architectural models.
2.2 The target: Acme architecture models

In our approach, the solution to our stated requirements is described using the Acme ADL [3]. It supports concepts such as Components, Connectors, Interfaces, Configurations and Rationale, as well as Ports, Roles, Properties and Representations (Figure 2). It includes a textual and a graphical language.

In the next section, we propose an approach to derive an early architectural design model from requirements models by using a model transformation approach.

3. SYSTEMATIC PROCESS TO ARCHITECTURAL DESIGN

In this section we present an approach for generating initial architectural models from i* requirements models. The approach is named Strategy for Transition between Requirements models and Architectural Models (STREAM) and includes a process with the following steps: (i) Prepare Requirements Models, (ii) Generate Architectural Solutions, (iii) Choose an Architectural Solution, and (iv) Derive Architecture. In order to perform steps (i) an (ii) horizontal and vertical model transformation rules, respectively, are proposed. Horizontal rules are applied to i* requirements models in order to increase their modularity and prepare them for the architectural design step. Vertical rules are developed to derive architectural models from the modularized i* requirements models. NFRs are used in step (iii). Depending on the quality attribute at hand, in step (iv) some architectural patterns may be applied. Each step will be presented as follows.

3.1 Prepare Requirements models

Often i* models are overloaded with information capturing features of both the system organizational environment and the software system itself. Therefore, the more detailed i* models are, the more complex they become. This rich ontology aligned with the common misuse of the decomposition mechanisms provided by the i*, produces i* models unnecessarily hard to read, understand, maintain and reuse. The first STREAM activity is concerned with the improvement of modularity of the expanded/refined system actor. This process allows delegation of different issues of a problem, initially concentrated into a single system actor, to new system actors, so that it is possible to deal with each of them separately. This activity uses: (i) a set of
metrics to evaluate the degree of modularity of initial and final i* models, (ii) a set of heuristics to guide the decomposition of the system actor, and (iii) a set of rules to transform i* models. Our approach is based in model transformations to assist the transition from requirements to architecture. This approach enables the definition of a systematic approach as well as contributes to keep traceability among software artifacts. Details on an earlier version of this activity can be found in [6]. We have now added a new rule, to deal with a special situation that may arise when independent sub-graphs, i.e., sub-graphs from different domains, have the same root goal. These sub-graphs are alternatives to satisfy this root goal. In this case, an actor is created for each alternative. Later, each one of these alternative actors in the architectural design will be considered as a different architectural solution. Note that elements from SR will be extracted to constitute a new actor.

Carrying out the Prepare Requirements models activity, the original model (Figure 1) is decomposed into more modularized system actors (Figure 3). In our example, there are two alternatives to achieve the Identification Details Collected goal (Figure 1). One relies on the use of standard forms, while a second alternative relies on encrypted forms. If we apply the horizontal rules (those that transform an initial i* model into a more modular one [6]) each alternative previously identified is moved to a different actor (see A1 and A2 dependencies in Figure 3). Thus, in our Medi@ example, we will have two SR models representing different configurations of the system and to be considered in the next activity. For the sake of space, here we present both alternatives in the same model. In fact, we should have two different SR models in order to represent each alternative.

### 3.2 Generate Architectural Solutions

Transformations of i* models into early Acme architectural models are conducted by vertical mapping rules. Hence, i* actors and dependencies are mapped to Acme elements. The rationale of this mapping follows the work presented in [7]. Figure 4 shows i* dependencies (without types) between actors (a) mapped into components and connectors in Acme graphical representation (b) and the corresponding Acme textual language (c).

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 [5]. An actor in i* is an active entity that carries out actions to achieve goals by exercising its knowhow [2]. An actor representing the software establishes a correspondence with modules or components [8]. 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 (Figure 4).

Connectors are architectural building blocks that regulate interactions among components [5]. 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 [7]. Thus, we can represent a dependency as an Acme connector. Ports and Roles are points of access that relate components and connectors (Figure 4-c).

However there are no ports in i*, but points where dependencies interact with actors. Depending on the direction of a dependency, we can know whether an actor is a depender or a dependee in the dependency. Hence, the roles of depender and dependee are mapped to the connector roles that are comprised by the connector (Figure 4-b). Thus, we can distinguish between required ports (when the actor is a dependee) and provided ports (when the actor is a dependee). For instance, Figure 4-c shows the use of property Required (Figure 4-c, line 4) and property Provided (Figure 4-c, line 7), indicating the direction of the communication between the DependerActor and DependeeActor components (Figure 4-b). Graphically, the required and provided ports are represented as white and black ports, respectively.
Thus, a component in Acme needs that another component carries out a service and the requisition of this service is done through a required port, while the result of this service is given by a provided port. As a result, 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.

A softgoal dependency is similar to a goal dependency but its fulfillment cannot be defined precisely. A softgoal is related to a NFR that will be treated by a task or a more specific softgoal. Hence, a softgoal dependency is mapped to a property with an enumerated type present into the port that plays the dependee role (provided port) of the connector. This enumerated type is used to describe the degree of satisfaction of the softgoal.

A task dependency represents that an actor depends on another one to execute a task [2] since a task describes or involves processing [8]. As a port in Acme corresponds to external interfaces of components and offer services. Hence, a task dependency is mapped directly to a provided port of the component that offers this port. In our Media example the Database component will have in its provided ports properties respectively associated to their tasks dependencies.

In a resource dependency, an actor depends on another actor to provide a type of resource (e.g., information). Therefore, a resource dependency is mapped to a return type of a property of a provided port. 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, this task is represented by a property of port. In our example, we do not have resource dependency between system actors.

The mapping rules can be applied to generate an early architectural solution in Acme. For example, an initial architectural model for the Media system is depicted in Figure 6. Each component mapped from an i* actor is renamed accordingly, taking into account the responsibility of each component. For example, the components could be named Statistics Producer, Database, MediaShop Interface, Media, and Details Identifier.

Note that MediaShop actor, that is the external entity interacting directly with the system (Figure 1), is also taken into account for the generation of the initial architecture. They represent interface components on architectural design. The MediaShop actor is mapped to the MediaShop Interface component (Figure 6).

Also note in the initial architectural model (see Figure 6) that the Media component aggregates in the same required port three connectors. This occurs because the same internal element of Media (Internet Shop Managed task of the Media actor in the requirements model), requires three services from Profile Manager. Moreover, to make the architectural design richer, we inserted labels in the connectors to indicate the nature of the relationship between components. For example, the connector between Statistics Producer and Media is labeled with produce statistics. Connectors labeled with adaptability, availability, adaptation and security, represent NFRs that are going to be achieved through provided ports offered by Profile Manager and Detailed Identifier components, respectively.

At the end of the Generate Architectural Solutions activity, we have two possible initial architectural solutions (Figure 6). These solutions are based on alternatives of i* models, one dealing with encrypted forms (A1) and another relying on the use of standard ones (A2). Moreover, other solutions could be generated based on architect’s expertise. Each of these solutions satisfies specific softgoals (NFRs) defined in the requirements model (Figure 3). Thus, these softgoals can influence the
decision about which architectural solution to use. This decision process constitutes the Choose an Architectural Solution activity.

### 3.3 Choose an Architectural Solution

After generating the early architectural solutions (Figure 6) based on the previous activities, the architect needs to decide the most appropriate one for a given context.

The softgoals present in the i* requirements models can be taken in consideration to support the design of a suitable architecture. Thus, this activity receives as input the modular requirements model and architectural solutions, both previously generated. The information related to softgoals present in requirements model are visualized in the connectors present in the generated architectural solutions.

In our Media example, we have two architectural solutions (see Figure 6). However we could have many more architectural solutions if we had different alternatives to achieve the same system function. Each solution tries to satisfy a specific set of softgoals. For example, the first A1 solution aims at fulfilling the security, availability and adaptability softgoals, whilst the second one (A2) targets the satisfaction of the availability and adaptability softgoals, but neglects the security softgoal, as seen in the requirements model (Figure 4). The next section shows how connectors representing softgoals could be refined though refinement patterns.

### 3.4 Derive Architecture

In order to fully achieve the chosen NFRs (softgoals), it is required to refine the architecture solution by using some architectural refinement patterns, such as the ones proposed by [9]. These patterns apply localized refinements to improve the architectural solution. The choice of a proper architectural refinement pattern depends on several factors, including the NFRs for the architectural solution.

Figure 7 shows an example of an architectural refinement pattern that satisfies the adaptability requirement. Adaptability is concerned with Detecting changes, Adapting and Monitoring changes (DAM). In this refinement pattern there are two components coupled by a connector. The connector shows that C1 provides services to achieve the Adaptability NFR required by C2. Adaptability is achieved by detecting and monitoring changes in the profile, context and environment of the user, and adapting the system to these changes. In many cases, these adaptability activities are not the main functionalities of C1. Thus, this pattern extracts the tasks related to adaptability from C1 to a new component named DAM. This is necessary to increase the cohesion of C1 and to concentrate adaptability related tasks into a separated component.

<table>
<thead>
<tr>
<th>Maintain Adaptability refinement pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Problem</strong></td>
</tr>
<tr>
<td>Context</td>
</tr>
<tr>
<td>C1</td>
</tr>
<tr>
<td>C1</td>
</tr>
</tbody>
</table>

![Figure 7. Architectural refinement pattern for achieve Adaptability NFR](image)

In the Medi@ example, adaptability is one of the NFRs that the architecture must meet. Both Media and MediaShop Interface require from Profile Manager adaption according to the user’s profile (see, in Figure 6, the adaptability connectors between Profile Manager and Media, and between Profile Manager and MediaShop Interface).

The refinement pattern is applied to locate the adaptability related tasks into the DAM component. Therefore, a new component is added in the architectural model to guarantee that the adaptability NFR is provided to Media and MediaShop Interface components. The Profile Manager component now provides information to the DAM component through a provided port, which replaces the two previous provided ports (Figure 6). Thus, in Figure 8 the DAM component is in charge to perform changes according to the user profile (Figure 8). As we aim at fulfilling the security, availability and adaptability NFRs, the second alternative for the component Details Identifier was chosen (Figure 8).

![Figure 8. Architectural Solution for Medi@ after application of pattern](image)

### 4. RELATED WORK

The i* modeling language is rich and expressive to describe system requirements [2]. However, approaches that use i* as the starting point of their methodologies, such as RISD [10] and Tropos [4], do not make a systematic transition to architecture. Therefore, the architect performs an ad hoc transition to the architectural design phase and do not benefit from all the expressiveness and richness present in the i* requirements model. Filling the gap between requirements and architectural design phases, will make the generated software systems closer to the requirements established by the stakeholders. We are working specifically with i* requirements models to produce architectural models in Acme. To the best of our knowledge, there are no studies in i* that generates architectural descriptions in Acme.

There are some other goal-oriented approaches that addressed the transition between requirements and architecture as, for example, SIRA [11], KAOS [9] and AOV-graph [12]. The SIRA approach [11] 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 [4]. Both requirements and architecture models are described in i*. Organizational architectural styles are chosen based on the NFRs [4]. Thus, an architectural model is created considering similarities of elements of requirements and of the chosen organizational style. Nevertheless, The SIRA approach [11] relies on i* to describe both the input (requirements model) and output (architectural model) models of the process.
Lamsweerde [9] 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, described using KAOS models, is generated from functional specifications. The resulting architecture is recursively refined to meet the various NFRs. KAOS models consist of a graphical tree and a formal language. Yet the generated architecture uses KAOS specifications instead of an ADL.

In [12] a set of mapping rules is proposed between the Aspectual oriented V-graph (AOV-graph) and the AspectualAcme, an aspect oriented ADL based on Acme. Each element (goal/softgoal/task) present in the AOV-graph is mapped to an element of the AspectualAcme, depending on the position of these elements in the graph hierarchy. The information about the source of each element, in the AOV-graph refinement tree, is registered in the properties of a component or a port in the architectural design. These properties make it possible to keep traceability and change propagation between AspectualAcme to AOV-graph models and vice-versa. Regrettably, it does not consider a pre-processing of the requirements models to improve modularity or an architectural reasoning in the context of requirements to address NFRs.

Last but not least, the mappings from i* to Acme constructs, defined in the STREAM approach, were based on the approach presented in [7]. However, we also consider refinement patterns.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we present the STREAM approach, whose goal is to generate an architectural model described in Acme from i* requirements model.

We proposed a process composed of four activities: Prepare requirements model, Generate Architectural Models, Choose an Architectural Solution and Derive Architecture. The first two activities are necessary to balance the responsibilities of a system actor, delegating them to other system actors. Thus, the requirements model is pre-processed to generate a more modular i* requirements model. To do so, a set of horizontal transformation rules was introduced in [6] and evolved in this work. The resulting requirements model is closer to an earlier architectural design, facilitating the transition from requirements to architectural design. From the modular i* model, we derived Acme architectural solutions through a set of mappings between the concepts of i* and Acme languages. Afterwards, one of the architectural solutions is chosen according to NFRs established in the requirements phase. If necessary, architectural refinement patterns [9] are chosen and applied. Therefore, these horizontal and vertical transformations involve steps before and after of change of notation. For example, in the transformation rules of i* often occurs division actors, whilst in the activity of refinement of the architecture, can occur grouping of components. Moreover, our mapping allows better traceability and propagation of changes. Furthermore, using a more general architectural language led us to propose more generic mapping rules, which can serve as a guide to derive architectural models in other ADLs. We illustrated the use of our approach using an e-commerce system as a running example.

Future works include: (i) Defining precisely both the horizontal and vertical rules using ATL (ATLAS Transformation Language) [13] and OCL (Object Constraint Language) [14]; (ii) Implementing the horizontal and vertical rules in a i* toolkit [15] based on model transformations; and (iii) performing some more experiments to validate the approach.

6. ACKNOWLEDGMENTS

This work was supported by several CNPq and CAPES research grants.

7. REFERENCES