The Journal of Systems and Software 85 (2012) 463–479

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Changing attitudes towards the generation of architectural models

Jaelson Castro[a], Marcia Lucena[b,*], Carla Silva[c], Fernanda Alencar[a], Emanuel Santos[a], João Pimentel[a]

[a] Universidade Federal de Pernambuco (UFPE), Recife, PE, Brazil
[b] Universidade Federal do Rio Grande do Norte (UFRN), Natal, RN, Brazil
[c] Universidade Federal da Paraíba (UFPB), Rio Tinto, PB, Brazil

ARTICLE INFO

Article history:
Received 8 June 2010
Received in revised form 23 May 2011
Accepted 24 May 2011
Available online 12 June 2011

Keywords:
Requirements engineering
Architectural design
Model driven development
Model transformations

ABSTRACT

Architectural design is an important activity, but the understanding of how it is related to requirements modeling is rather limited. It is worth noting that goal orientation is an increasingly recognized paradigm for eliciting, modeling, specifying, and analyzing software requirements. However, it is not clear how goal models are related to architectural models. In this paper we present an approach based on model transformations to derive architectural structural specifications from system goals. The source and target languages are respectively the i* (iStar) modeling language and the Acme architectural description language. A real case study is used to show the feasibility of our approach.

1. Introduction

Requirements engineering (RE) (Kotonya and Sommerville, 1998) and software architecture design (SAD) (Hofmeister et al., 2001) are initial activities of a software development process. Since current software systems present increasing complexity, diversity and longevity, their development must consider the use of proper methods and modeling languages both for RE and SAD. A great challenge, in this context, is the development of systematic methods for designing architectures that satisfy requirements specifications. Some efforts have been made to understand the interaction between RE and SAD activities (Berry et al., 2003; Castro and Kramer, 2001). In fact, with the widely use of iterative and incremental software development processes as the *de facto* standard, a strong integration between RE and SAD activities can facilitate traceability and the propagation of changes between the models produced in these activities (Silva et al., 2007). In this context, we can highlight the twin peaks model (Nuseibeh, 2001), which emphasizes the co-development of requirements specification and architectural design description, incrementally elaborating details in both artifacts. Recognizing the close relation between architectural design description and requirements specification, as argued in (de Boer and van Vliet, 2009), using model transformation approaches (Czarnecki and Helsen, 2003) appears as an effective way to generate architectural models from requirements models, in which the correlation between requirements and architectural models can be accurately specified.

This paper presents a systematic process based on model transformations to generate architectural models from requirements models and includes horizontal and vertical transformations rules. The horizontal transformations are applied to the requirements models resulting in intermediary requirements models closer to architectural models (Lucena et al., 2009a). Vertical transformations map these intermediary models into architectural models (Lucena et al., 2009b). The activities related to architectural design involves the selection and application of architectural patterns that best satisfy non-functional requirements.

In our approach, requirements models are described using the i* (iStar) (Yu, 1995), a goal oriented modeling language defined in terms of strategic actors and social dependencies among them. Whereas architectural models are described using the Acme ADL (Garlan et al., 1997), which provides a simple structural framework for representing architectures.

Our choice of i* for specifications was motivated by the growing number of groups around the world that have been using the i* modeling framework in their research on early requirements engineering, business process design, organization modeling, software development methodologies, and more (Yu et al., 2011; Castro et al., 2010; Franch, 2010). On the other hand, for architecture we opt for Acme because it is a generic ADL that can be used as a common interchange format for architecture design tools and/or as a foundation for developing new architectural design and analysis tools.

* Corresponding author. Tel.: +55 84 32153814x218.
E-mail addresses: jbc@cin.ufpe.br (J. Castro), marciaj@dimap.ufrn.br (M. Lucena), carla@dce.ufpb.br (C. Silva), fernanda.ralencar@ufpe.br (F. Alencar), ebs@cin.ufpe.br (E. Santos), jhcp@cin.ufpe.br (J. Pimentel).

0164-1212/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2011.05.047

Note that, if required, Acme descriptions can be mapped to UML 2.0 (Goulão and Abreu, 2003).

The rest of this paper is organized as follows. Section 2 defines our problem statement. Section 3 introduces our case study and gives an overview of the main concepts of the i* and Acme languages. Section 4 outlines our approach, applied to the case study. Section 5 summarizes our work and points out open issues and related works. Last but not least, Section 6 presents the conclusions and future works.

2. Problem statement

Many researchers associate requirements with the problem-space and architecture with the solution-space (de Boer and van Vliet, 2009). This leads to a semantic gap between requirements specifications and architectural design descriptions, which consists of conceptual differences between what to do (requirements) in opposition to how to do it (architecture, design and coding). From the challenges that appear when developers try to match requirements and architecture, presented in (Grünbacher et al., 2004), we highlight:

i. Requirements are often captured informally in natural language while architecture is specified in a semi-formal way.
ii. Properties of the software described as non-functional requirements (NFRs) are difficult to specify in architectural models.
iii. The iterative and concurrent evolution of requirements and architectural models are usually based on incomplete requirements. Besides, some requirements can only be understood/elicited after modeling the architectural design.
iv. Mapping requirements specifications to architectural design descriptions and promoting a consistent maintenance of these artifacts are difficult tasks. Moreover, a simple requirement may be addressed by various architectural elements and a single architectural element may have non-trivial links to several requirements.
v. In realworld, large-scale systems have to satisfy thousands of requirements. For developing this kind of system, it is difficult to identify and refine the architecturally relevant information contained in the requirements, such as NFRs.

Despite these challenges, it is unusual to have a systematic approach that is concerned with analyzing and understanding the requirements and producing a suitable set of architectural solutions that satisfactorily meet the requirements. The techniques and methods used for software development should include a way of systematically dealing with the relationship between requirements models and architectural models.

What is the point of building rich, consistent and complete requirements models if the transition to the architecture is developed on an ad hoc basis, without paying attention to the relationship between the consumed and produced models? Moreover, some information present in the requirements specification, such as the NFRs, can be lost in the transition to the architectural design, since the operationalization of these NFRs are often not considered by current architectural design methods. In summary, it is important to ensure that the requirements information is not lost during the development process.

3. Background

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

3.1. BTW project

The BTW-UFPE project (Pimentel et al., 2010), presented in the SCORE contest held at ICSE 2009 (SCORE, 2009), is used to illustrate our approach. BTW (By The Way) consists in a route-planning system that helps users through advices about a specific route searched by the user. This information is posted by other users and might be filtered to provide for the user only relevant information about the place that he/she intends to visit.

The BTW-UFPE team generated artifacts that included i* requirement models. We chose this project for two reasons: (i) it is a real case study that resulted in an awarded software system, and (ii) the produced i* models are not large but have enough complexity to illustrate the benefits of our approach. Figs. 1 and 2 show i* models for the BTW project. Its complete models can be found in (Pimentel et al., 2010).

3.2. The source: i* goal model

i* (iStar) defines models to describe both the system and its environment in terms of intentional dependencies among strategic actors (Yu, 1995). There are two different models: the strategic dependency (SD) describes information about dependencies and the strategic rationale (SR) defines actor details.

The SR model complements the information provided by the SD model by exploiting internal details of their strategic actors to describe how the dependencies are accomplished. For example, Fig. 1 presents the SD model of the BTW project, focusing on dependencies among actors. In i* models, a depending actor is called a depender, and an actor that is depended upon is a dependee. Thus, in Fig. 1 there is a software actor (*BTW*), actors representing human agents (*Travelers*, that can be *Advice Giver* and *Advice Receiver*), and an actor representing an external system (*Internet Provider*).

Fig. 2(a) is a SR model showing the BTW actor and its internal details. Fig. 2(b), for instance, presents a partial view of the dependencies between *Advice Giver* and *BTW* actors. *BTW* represents the software system to be developed. Considering the *Precise Information* softgoal dependency in Fig. 1, the *BTW* actor is the depender actor whereas the *Advice Giver* actor is the dependee actor of this dependency. Thus, there are four types of dependencies, according to the dependum—i.e., according to what the depender requires from the dependee. The dependum can be a softgoal, a goal, a task or a resource (Fig. 1). Softgoals are generally used to describe the actors' desires related to quality attributes of their goals. The tasks represent a way to perform some activity, that is, they show how to perform some action to obtain satisfaction of a goal or softgoal. The resources represent data or information that an actor may provide or receive. Each type of dependency has a different meaning, according to the definition of the respective dependum. A goal dependency states that the depender needs the dependee to satisfy a goal for him. Similarly, in a softgoal dependency the depender needs the dependee to meet a softgoal. In a task dependency, the dependee is asked to perform an activity for the depender. A resource dependency express that the depender needs some resource that may be provided by the dependee. 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 NFRs, while goals, tasks and resources are associated to system functionalities (Yu et al., 2008).

Actor's internal details also include tasks, goals, resources and softgoals, which are further refined using task-decomposition, means-end and contribution links (Fig. 2b). 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

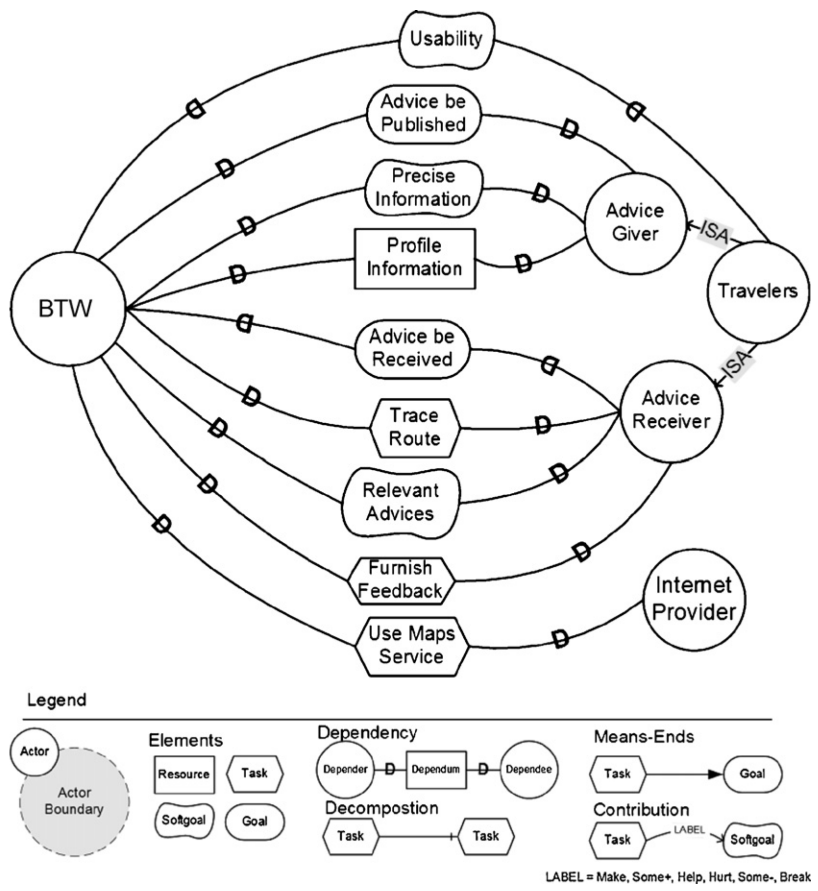


Fig. 1. SD model of BTW project with BTW Actor and dependencies.

intentional element (e.g., relationship between the *Select Advice by User History* task and the *Relevant Advice be Chosen* 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). Note that an element can be shared by different sub-graphs (through task-decomposition, contribution or means-end link). For example, in Fig. 2a, the *Select Placemark* task is shared through task-decomposition by both the *Provide Maps Services* and *Add Advice Content* tasks.

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

3.3. The target: Acme architecture models

A set of elements is important when describing instances of architectural designs. According to Taylor et al. (2009), these elements include components, connectors, interfaces, configurations and rationale. Acme ADL (Garlan et al., 1997) supports each of these concepts but also adds ports, roles, properties, and representations. Besides, Acme has a graphical (Fig. 3a) and a textual (Fig. 3b) language. Acme components represent computational units of a system (e.g., *BTW*, Fig. 3). Connectors represent and mediate interactions between components (e.g., *Publish_In_Map*, Fig. 3).

A connector can present a label that represents the connector's identifier. 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. Attachments are used to link ports and roles (e.g., *port0* of *BTW* component is linked with *role1* of connector *Publish_In_Map*).

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 connectors whose edges represent their interconnectivity. Properties are annotations that define additional information about elements (components, connectors, ports, roles, and systems).

Representations (e.g., *Map_Info_Publisher*, Fig. 3) allow the description of details about the sub-architecture that refines a component, a connector, a port or a role. By using the binding mechanism, the ports of a sub-component can be linked to the correspondent ports of a parent component. 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 (Taylor et al., 2009).

Architectural design is not a trivial task. It depends on the expertise of the architects and on how they understand the requirements. To make this task more systematic, we propose a model transformation approach to derive an early architectural design from requirements models. In a software development process, software architecture models should be related to requirements models.

The relationship between them is usually established in an ad hoc manner, relying on the expertise of the software architect. A challenge, when using model transformation approaches, is to define the transformation rules that guide the software developer through the process. In this paper, we propose a process that smoothly transforms the initial requirements model into an early

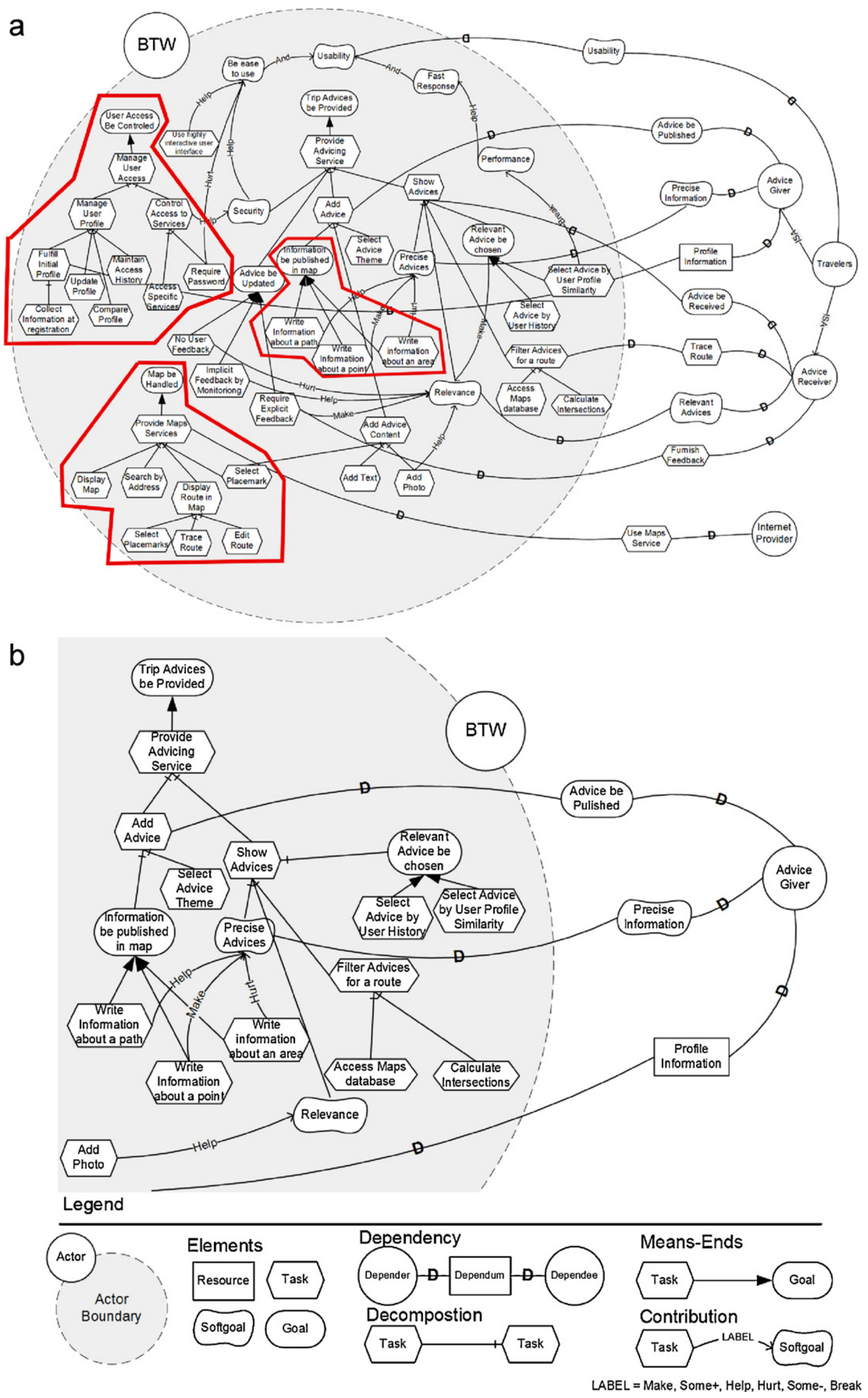
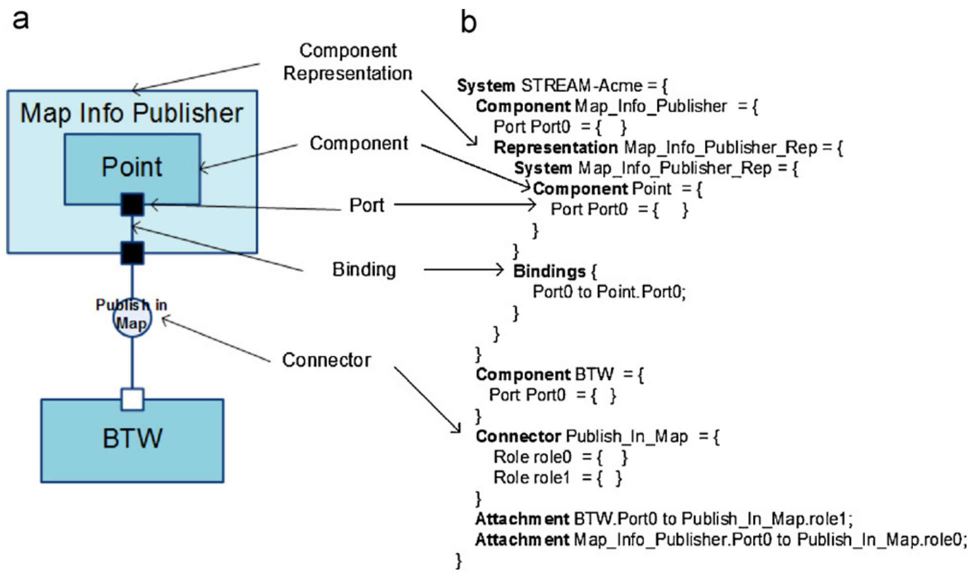


Fig. 2. BTW strategic rationale model with dependencies (a) and partial BTW strategic rationale model (b).



```
System STREAM-Acme = {
  Component Map_Info_Publisher  = {
    Port Port0  = {   }
    Representation Map_Info_Publisher_Rep = {
      System Map_Info_Publisher_Rep = {
        Component Point  = {
          Port Port0  = {    }
        }
      }
      Bindings {
        Port0 to Point.Port0;
      }
    }
  }
  Component BTW  = {
    Port Port0  = {  }
  }
  Connector Publish_In_Map  = {
    Role role0  = {   }
    Role role1  = {  }
  }
  Attachment BTW.Port0 to Publish_In_Map.role1;
  Attachment Map_Info_Publisher.Port0 to Publish_In_Map.role0;
}
```

Fig. 3. Overview of the Acme graphical and textual notation.

architectural model. This process will be presented in the next section.

4. Stream process

Our approach was named “Strategy for Transition between REquirements models and Architectural Models” (STREAM). It consists of: (a) a set of transformation rules to prepare requirements models; (b) a set of transformation rules to generate architectural models; and (c) a systematic process to assist developers. The process includes four main macro-activities that will be detailed in the next sections: (i) prepare requirements models, (ii) generate architectural solutions, (iii) select an architectural solution, and (iv) refine architecture. Fig. 4 presents the flow of these macro-activities using the SPEM notation (Consortium, 2006). Each macro-activity aggregates a flow of activities that will be detailed afterwards.

The prepare requirements models activity (i), in particular, describes 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. It is worth noting that preliminary versions of the prepare requirements models (i) and generate architectural solutions (ii) activities have already been presented respectively in Lucena et al. (2009a,b).

4.1. Prepare requirements

i* offers expressive models to capture social and intentional characteristics of a system organizational context. Actors and dependencies are core concepts. Actors have desires and needs. They are concerned with opportunities and vulnerabilities. Thus, i* is used to explicitly capture stakeholders’ motivations and rationale in a requirements model (Yu, 1995).

However, i* models are often overloaded with information that captures features of both the system organizational environment and the software system itself. The more detailed i* models are, the

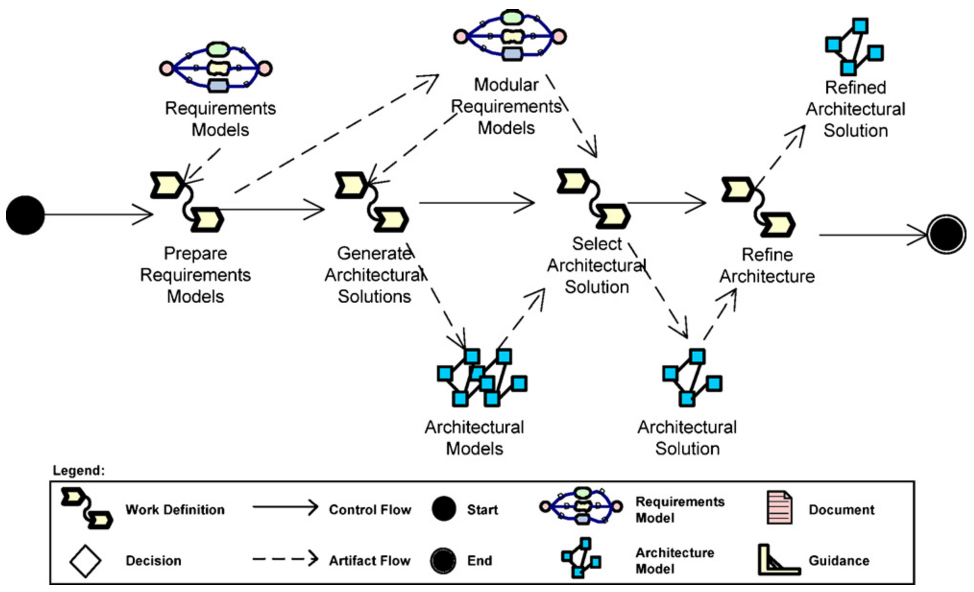


Fig. 4. Overview of the STREAM process.

more complex they become. Hence, i* models can become unnecessarily hard to read, understand, maintain and reuse.

Accordingly to Estrada et al. (2006), the i* modeling language can be evaluated using several aspects, from which we highlight modularity management. Modularity measures the degree to which the modeling language offers well-defined building blocks (i.e., modules) for building a model. The modules should allow the encapsulation of internal structures of the model in a concrete modeling construct. This characteristic ensures that changes in one part of the model will not be propagated to other parts, whereas complexity management measures the capability of the modeling method to provide a hierarchical structure for its models, constructs and concepts.

Although i* incorporates a decomposition mechanism based on strategic actors, which could be used to improve modularization of i* models, often the way in which this mechanism is deployed is not suitable to produce models that are easy to maintain and reuse. Current methods for i* modeling, represent the rationale of an actor in a monolithic way. Sometimes several refinements are described in a mixed way, making it hard to visualize the boundaries of sub-graphs related to specific domains. This poor modularity compromise the management of the complexity of the models, an important pre-requisite for the adoption of i* in industrial settings (Estrada et al., 2006).

Considering this disadvantage of i * with respect to modularity, there are some works addressing this issue. The most remarkable contributions are related to incorporating of aspects (Alencar et al., 2010), services (Estrada, 2008) and modules (Franch, 2010) into i*. Among these works, the most closely with the modular approach used in our approach is the work that incorporates modules (Franch, 2010), specifically the concept of SR module. An SR module is composed of SR elements and links among them.

It should contain at least two SR elements, with a minimum of one link among them. At least one of the SR elements shall be a root. From the root elements, all other intentional elements shall be reachable.

Therefore, we propose to take the modularity problem by means of a divide and conquer strategy, whereas a strategic actor can be used as a decomposition mechanism that divides complex actors into meaningful and manageable sub-actors. We claim that such pre-processing can greatly improve the modularity of i* models.

The aim of this activity is to improve the modularity of the expanded/refined software actor. It allows delegating different issues of a problem, initially concentrated into a single actor, to new actors. This enables to deal with each of them separately, following the separation of concerns principle (Dijkstra, 1976). Thus, this activity uses as decomposition criterion the separation and modularization of elements that are not strongly related to the application domain and that could easily be used in other domains.

In order to assist the requirements engineer to identify the elements that can be removed from the software actor, we propose the following heuristics: (H1) Search internal elements in the software actor that are independent of the application domain; (H2) Verify whether these elements can be moved from the software actor to another software actor without compromising the behavior and the understandability of the actor's internal details; (H3) Check whether these elements can be reused in different domains.

For example, in the BTW SR model (see the selected parts of Fig. 2a), which captures the web recommendation system requirements (Pimentel et al., 2010), we can identify some elements that are not fully related to the current application domain (recommendation systems). Furthermore, they can be delegated to other actors, which could possibly be reused in other applications. Note that this analysis is performed by the requirements engineer.

In our case study, some sub-graphs internal to the BTW actor are considered independent from the recommendation application domain. Therefore, they can be moved to new software actors. Thus, sorting out the independent elements into other actors can improve reusability and maintainability of the software requirements specification. In fact, considering the BTW SR model (see the selected parts of Fig. 2a), the following elements could be seen as being independent of the application domain: *Map to be Handled*, *User Access be Controlled* and *Information be published in map*.

The decomposition has the objective of modularizing i* models by delegating responsibilities of the software actor to other actors that are in charge of a particular concern. In this paper, concerns are cohesive groups of domain independent elements. The modularization is performed by a set of horizontal transformations rules. Each rule performs a small and localized transformation that produces a new model that decomposes the original model. Both the original and the produced models are described as i* models.

The proper rule to be applied depends on the type of relationship (task-decomposition, means-end and contribution) between the elements to be moved out and the elements that will remain in the original software actor. This delegation establishes a dependency relationship between the new actors and the original actor, aiming at keeping the same semantics of the original model in the resulting model. A special case to be considered is when some element is shared (through task-decomposition, means-end or contribution relationship).

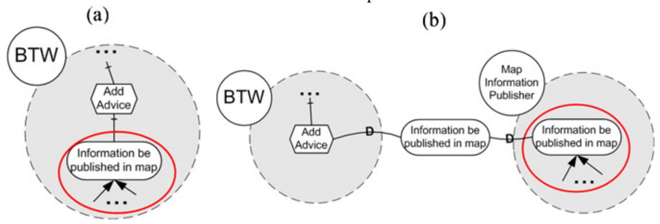
Hence, we propose four horizontal transformation rules: HTR1, HTR2, HTR3 and HTR4. The first three rules address, respectively, elements that participate in task-decomposition, means-end or contribution relationship. The last one deals with situations where there is some shared element.

Horizontal transformation rule 1 (HTR1) is a transformation rule that moves a sub-task, present in a task-decomposition, to another actor. Table 1 presents the structure of HTR1, showing two models, one to illustrate the context in which the original model can match before applying the rule (a) and other illustrating the resulting model after applying the rule (b). In addition, a description of the conditions required for the application of the rule (c) and the effects produced by the rule (d). Since HTR1 and HTR3 were applied in BTW application (Fig. 2), so only these rules are used to illustrate the models in (a) and (b). In order to illustrate the remaining rules HTR2 and HTR4 another example, based in a web application (Castro et al., 2002), was used.

Horizontal transformation rule 2 (HTR2) considers the situation where the sub-graph to be moved has the root element as a "means" in a means-end relationship (see Table 2).

Table 1
Transformation rule to move sub-element in a task-decomposition link.

HTR1 – Move a sub-element in a task-decomposition

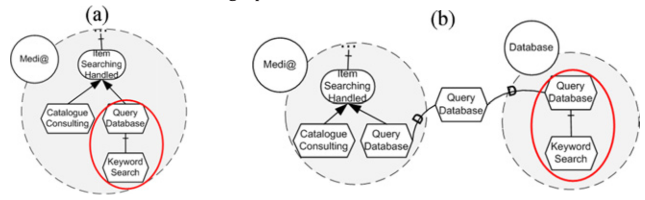

(a)
(b)
BTW
Add Advice
Information be published in map
BTW
Add Advice
D
Information be published in map
D
Map Information Publisher
Information be published in map

(c) Pre-condition: a root element (task or goal) of the graph to be transferred is decomposed into sub-elements that do not share any sub-element through task-decomposition or means-end links.

(d) Effects: the sub-graph that is independent from the application domain (e.g., the sub-graph whose root is the *Information be published in Map* goal) is moved from the original actor to a new actor. The sub-element that was moved out, but remain linked to the original actor (e.g., *Information be published in Map* goal), will be replicated as a dependency relationship of same type relating to the original actor, as the depender, and the new actor, as the dependee. Besides, all the existent external dependencies of this transferred sub-graph also will be transferred to the new actor.

Table 2
Transformation rule to move the“means” sub-graph in a means-end link.

HTR2 – Move “means” sub-graph in a means-end link



(c) Pre-condition: a root goal of a graph (e.g., *Item Searching Handled*) is an "end" in one or more means-end relationships and, at least one of its "means" is a task (e.g., *Query Database*). Besides, its sub-graph does not share any element (through task-decomposition or means-end links) with other sub-graphs (independent sub-graph).
(d) Effects: an independent sub-graph (alternative) will be moved to a new actor. The root of transferred sub-graphs (e.g., *Query Database* task) must be replicated inside the original actor. From the replicated elements, a new dependency of the same type and name must be created from the original actor to the sub-graph's root moved to the new actor.

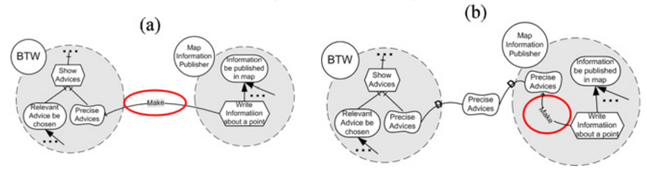
Horizontal transformation rule 3 (HTR3) is a restrictive rule, since after applying rules HTR1 and HTR2, the resulting model may not be in conformity with the i* syntax. This rule was defined to preserve the information about contribution links and coherence with the i* syntax (Table 3).

On the other hand, horizontal transformation rule 4 (HTR4) addresses situations involving elements that participate in different sub-graphs (Table 4). Thus, HTR4 is applied when the sub-graph to be moved has a shared sub-element with other sub-graph (see *Get Item Detail* in Table 4a). This rule suggests that a priority policy is established to define where the shared element will remain (e.g., *Get Item Detail* task, Table 4a). Thus, it is necessary to check the types of the root elements in the sub-graphs that share the element. The shared elements will remain with the sub-graph that has the root with type of highest priority.

In our study, after applying the horizontal rules to the selected elements, three new actors were created to modularize the following sub-graphs: *Map be Handled*, *User Access be Controlled* and *Information be published in Map* goals. Fig. 5a and b show, respectively, the SD model and part of the SR model of the BTW actor after applying the horizontal rules. As these new actors have names based on the name of their root elements, we suggest using a specific name related to the domain addressed by these elements. Thus, we have respectively *Mapping Handler*, *User Access Controller*, and *Map Information Publisher*.

Table 3
Transformation Rule to move a contribution link crossing the actor's boundary.

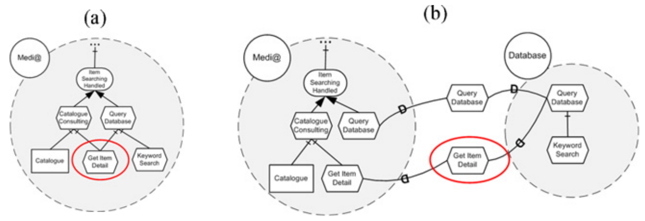
HTR3 – Contribution link crossing actor's boundary


(a)
(b)
BTW
Show Advices
Relevant Advice be chosen
Precise Advices
Make
Map Information Publisher
Information be published in map
Write Information about a point

(c) Pre-condition: there are elements such as task, goal or softgoal contributing to the achievement of softgoals that are inside of another actor's boundary (e.g., *Write Information about a point* task contributes positively to Precise Advices softgoal).
(d) Effects: a softgoal element, with the same name, must be created inside the actor from where the contribution link goes out (e.g., *Map Information Publisher* actor). The contribution link is kept inside that actor. A softgoal dependency, with the same name, must be created from the softgoal element present in the original actor (e.g., *Precise Advice* softgoal inside *BTW*) to the softgoal created in the other actor (e.g., *Precise Advice* softgoal inside *Map Information Publisher*).

Table 4
Transformation rule to move shared elements.

HTR4 – Move a shared sub-element



(c) Pre-condition: There is an element (*Get Item Detail* task) that is shared by different sub-graphs (through task-decomposition, contribution or means-end link) and at least one of these sub-graphs is to be moved to a new actor (after applying rules HTR1 or HTR2).
(d) Effects: The shared element will remain in the sub-graph whose root element has the highest priority, according to this order: goal, softgoal and task. Case all sub-graphs' roots have the same type, verify the hierarchical position of each root of the sub-graphs that share the element check the root that is closer to the most general root will keep the shared element. Case all the roots of the sub-graphs have the same type and are on the same hierarchical level in relation to the most general root, then the shared element will stay with the sub-graph that remains in the original actor.
The relationships with the elements that will remain in the original actor are to be replaced by dependencies (e.g., *Get Item Detail* dependencies), as stated by the rules HTR1, HTR2 and HTR3.

As the horizontal rules are applied, the i* model is transformed into an intermediary model closer to an early architectural design. This intermediary model is used as an input to the *Generate Architectural Solutions* macro-activity. All the new created actors, as well as the main software actor, the actors representing human agents (*Advice Giver* and *Advice Receiver*), the *Internet Provider* actor, and the dependencies among each other will be used by the vertical transformation rules. Next section presents the rationale behind the vertical transformation rules to produce Acme architectural models from more modular i* requirements models.

4.2. Generate architectural solutions

The activities of a software development process create models using specific model languages. In a model transformation approach, new models are generated by transforming a previous model. Therefore, how models are generated from a stage to another rely on how the transformation rules are defined considering the main elements of each involved modeling language. In the *Prepare Requirements Models* activity we used horizontal rules because both models, original and final, were in the same abstraction level (i* models). In the *Generate Architectural Solutions* activity, vertical rules are used because they translate models of different abstraction levels, i.e., from i* models into Acme models.

In this second activity, we start by establishing the vertical transformation rules considering only actors and dependencies to map i* elements to specific elements of an ADL, similarly to the approach presented in Castro et al. (2003). Thus, Table 5 shows a generic mapping while Tables 6–9 show mappings according to type of dependency, respectively goal, task, resource and softgoal.

Table 5 shows a generic mapping of actors and dependencies without considering the dependency type, in i* (Table 5a), to components and connectors in Acme graphical (Table 5b) and textual language (Table 5c).

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 (Taylor et al., 2009). An actor in i* is an active entity that carries out actions to achieve goals by exercising its knowhow (Yu, 1995). The actor representing the software establishes a correspondence with modules or components (Grau

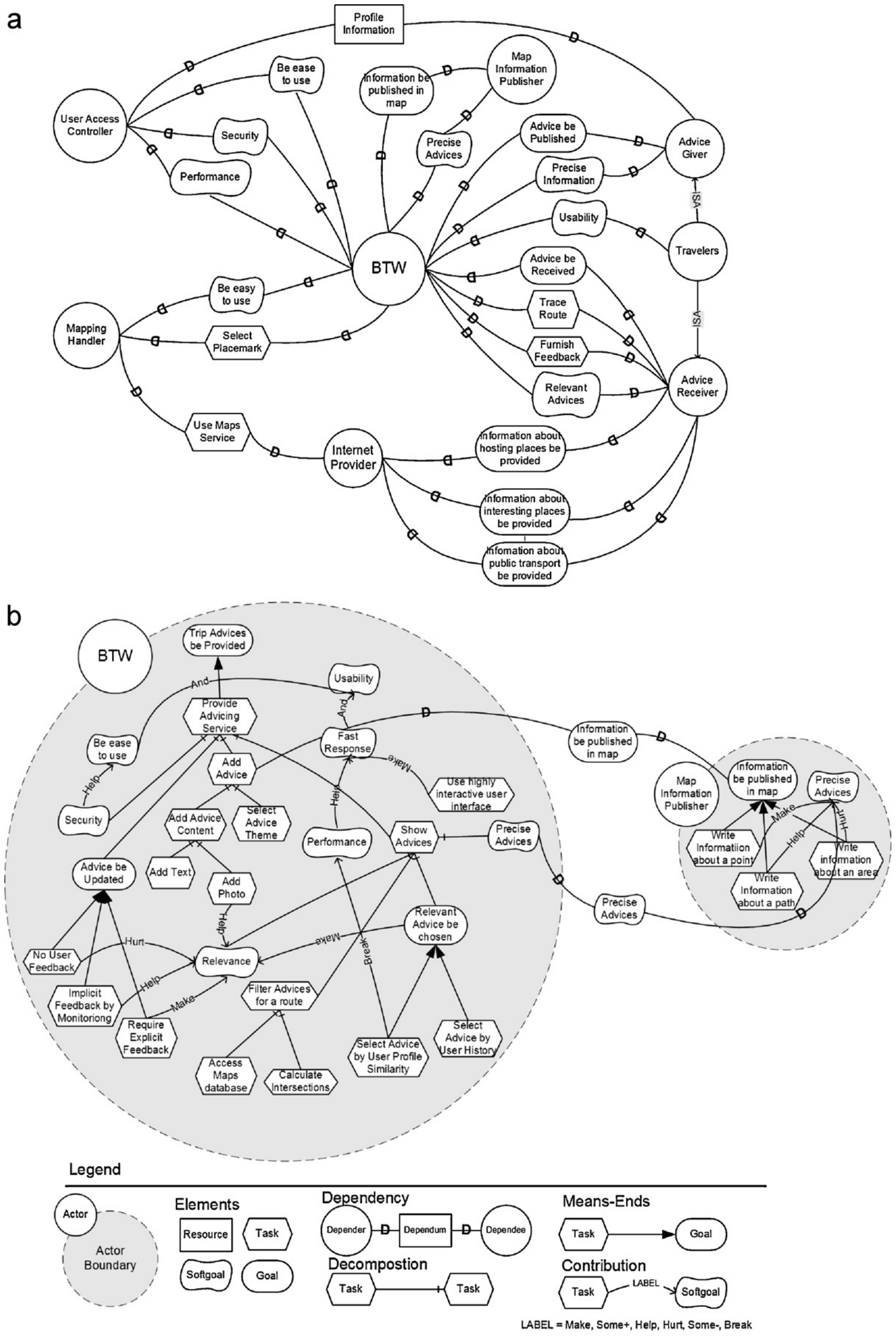
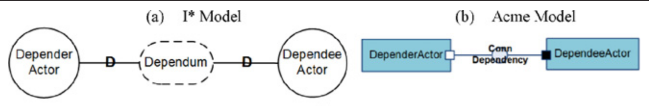


Fig. 5. Modular i* model: SD model (a) and excerpt from SR model (b).

Table 5
Mapping a generic dependency between i* actors to Acme.



(c)

```
1 SystemNameSystem= {
2     ComponentDependerActor   = {
3         Port port1   = {
4             Property Required : boolean = true;         }   }
5     ComponentDependeeActor   = {
6         Port port2   = {
7             Property Provided : boolean = true;         }    }
8     ConnectorConnDependency   = {
9         Roledepender   = {     }
10        Roledependee   = {     }     }
11     Attachment DependerActor.port1 toconnDependency.depender;
12     Attachment DependeeActor.port2 toConnDependency.dependee;}
```

and Franch, 2007). In addition, an actor may have as many interactions points as needed. Hence, a software actor in i* can be represented as a component in Acme (Table 5). If the actor being mapped represents a human agent, it will be mapped to an Acme component that represents the communication interface, or point of interaction, between the human agent and the software actors (e.g., *Advice Giver* and *Advice Receiver* in Fig. 6).

In Acme, a component requires that another component carries out a service and the requisition of this service is performed through a required port, while the result of this service is done through a provided port. Thus, a connector allows the communication between these ports (Garlan et al., 1997). Connectors are architectural building blocks that regulate interactions among components (Taylor et al., 2009) and mediate the communication and coordination of activities among components. Note that a label (graphically represented as a circle) can be attached to a connect or to represent its identifier (see Table 5b).

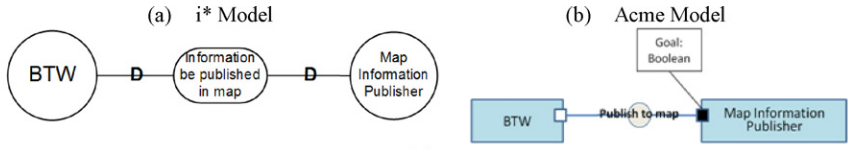
In i*, a dependency describes an agreement between two actors playing the roles of depender and dependee (Castro et al., 2003). Thus, we can represent an i* dependency as an Acme connector. The label of the connector can be mapped accordingly (see Table 5c, line 8). Ports are external interfaces that acts as access points among components and connectors. The concept of port is not present in i*. However, there are points where dependencies interact with actors and define if an actor plays the role of a depender or a dependee in a dependency, according to the direction of the dependency. Hence, the roles of depender and dependee are mapped to roles that are comprised by the connector (Table 5c). We can distinguish between required ports (mapped from the depender actor) and provided ports (mapped from the dependee actor). For instance, Table 5 shows the use of the Required property set with "true" value (Table 5c, line 4) and the Provided property set with "true" value (Table 5c, line 7) indicating the communication direction between the DependerActor and DependeeActor components. In i*, a depender actor depends on a dependee actor to accomplish a type of dependency. Therefore, a component offers services to another component using its provided ports and requires services from another component using its required ports. Thus, in a port, a Required property is true if the port is in charge of requesting a service while a Provided property is true if the port is in charge of providing a service (Table 5, lines 4 and 7).

After applying this mapping to the software actors of the BTW requirements model (Fig. 4), four components will be generated from them: *BTW*, *Mapping Handler*, *User Access Controller* and *Map Info Publisher*. Besides, two components will be generated from the actors representing human agents and one component will be generated from the Internet Provider actor. Each dependency is mapped to a connector and the roles of each connector's end (depender or dependee) will be defined according to the dependency direction. Thus, the roles of depender and dependee are mapped to connector roles that are comprised by the connector. In addition, required ports (inside the component mapped from the depender actor) and provided ports (inside the component mapped from the dependee actor) are presented. Thus, when an actor has at least one dependency as a dependee, its equivalent component will have at least one provided port (Table 5c, line 7). For example, the *Mapping Handler* component will have a provided port for the *Placemark* resource dependency. Having all components, ports, connectors and roles mapped and defined, the next step is to analyze each type of dependency.

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 the connector and in the ports that play the connector roles.

A goal dependency is mapped to a Boolean property inside a provided port responsible for providing a service that will fulfill the goal (Table 6c, line 8). The goal property is Boolean to indicate whether the component has accomplished the goal by providing a service. Initially, the value of this property is false indicating that the service has not been offered yet. Its value becomes true when the component actually provides the required service. Such goals pre-

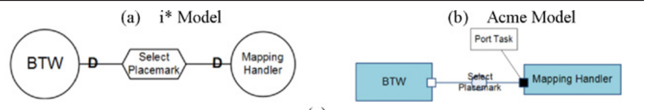
Table 6
Mapping a goal dependency to Acme.


(a) i* Model
BTW
D
Information be published in map
D
Map Information Publisher
(b) Acme Model
Goal: Boolean
BTW
Publish to map
Map Information Publisher

(c)

```
1 SystemRoute-planning= {
2    Component BTW   = {
3        Port port1   = {
4            Property Required : boolean = true;         }   }
5    ComponentMap_Information_Publisher   = {
6        Port port2   = {
7            Property Provided : boolean = true;
8            Property goal: Boolean = false;        }    }
9    ConnectorPublish_in_Map = {
10       Roledepender   = {     }
11       Roledependee   = {     }      }
12   Attachment BTW.port1 toPublish_in_Map.depender;
13   Attachment Map_Information_Publish.port2 to
Publish_in_Map.dependee;}
```

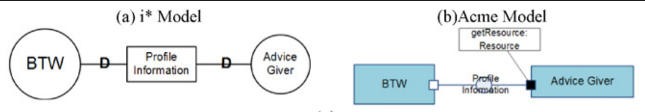
Table 7
Mapping a task dependency to Acme.



(c)

```
SystemRoute-planning= {
    Component BTW   = {
        Port port1   = {
            Property Required : boolean = true;          }   }
    ComponentMapping_Handler   = {
        Port Task   = {
            Property Provided : boolean = true;         }    }
    ConnectorSelect_Placemark = {
        Roledepender   = {     }
        Roledependee   = {     }     }
    Attachment BTW.port1 toSelect_Placemark.depender;
    AttachmentMapping_Handler.Taskto Select_Placemark.dependee;}
```

Table 8
Mapping a resource dependency to Acme.



(c)

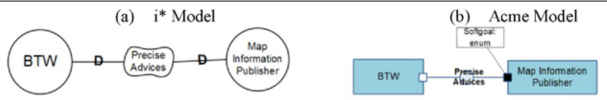
```
SystemRoute-Planning= {
    Component BTW   = {
        Port port1   = {
            Property Required : boolean = true;          }   }
    Property Type Resource;
    ComponentAdvice_Giver   = {
        Port port2   = {
            Property Provided : boolean = true;
            Property getResource: Resource;    }    }
    ConnectorProfile_Information = {
        Roledepender   = {     }
        Roledependee   = {     }     }
    Attachment BTW.port1 toProfile_Information.depender;
    AttachmentAdvice_Giver.port2to Profile_Information.dependee;}
```

scribe the intended system behaviors declaratively. A behavioral goal implicitly defines a maximal set of admissible system behaviors. Goals describe functions the system will perform. Moreover, they have a well defined criteria for satisfaction.

Observe that in this case, the type of property is Boolean in order to represent the goal fulfillment (true) or not (false). Applying this mapping to the BTW case study implies that the *MapInformation Publisher* component will add a new Boolean property in the provided port related to the *Information be published in map* goal dependency (*Publish in Map* connector). A note (e.g., Table 6b) is attached to a port to highlight added elements used in textual notation (e.g., Table 6c, line 8).

A task dependency represents that an actor depends on another to execute a task (Yu, 1995) and that a task describes or involves processing (Grau and Franch, 2007). As explained before, a port in Acme corresponds to an external interface of the component and offers services. Hence, a task dependency is mapped directly to a provided port of the component that offers the ser-

Table 9
Mapping a softgoal dependency to Acme.



(c)

```
SystemRoute-Planning= {
    Component BTW   = {
        Port port1   = {
            Property Required : boolean = true;          }   }
    Property Type DegreeOfSatisfactionOfSoftgoal = enum {Satisfied,
Partially Satisfied, Conflict, Unknown, Partially Denied, and Denied};
    ComponentMap_Information_Publisher = {
        Port port2   = {
            Property Provided : boolean = true;
            Property softgoal: DegreeOfSatisfactionOfSoftgoal;    }    }
    ConnectorPrecise_Advices = {
        Roledepender   = {     }
        Roledependee   = {     }     }
    Attachment BTW.port1 toPrecise_Advices.depender;
    AttachmentMap_Information_Publisher.port2toPrecise_Advices.dependee;}
```

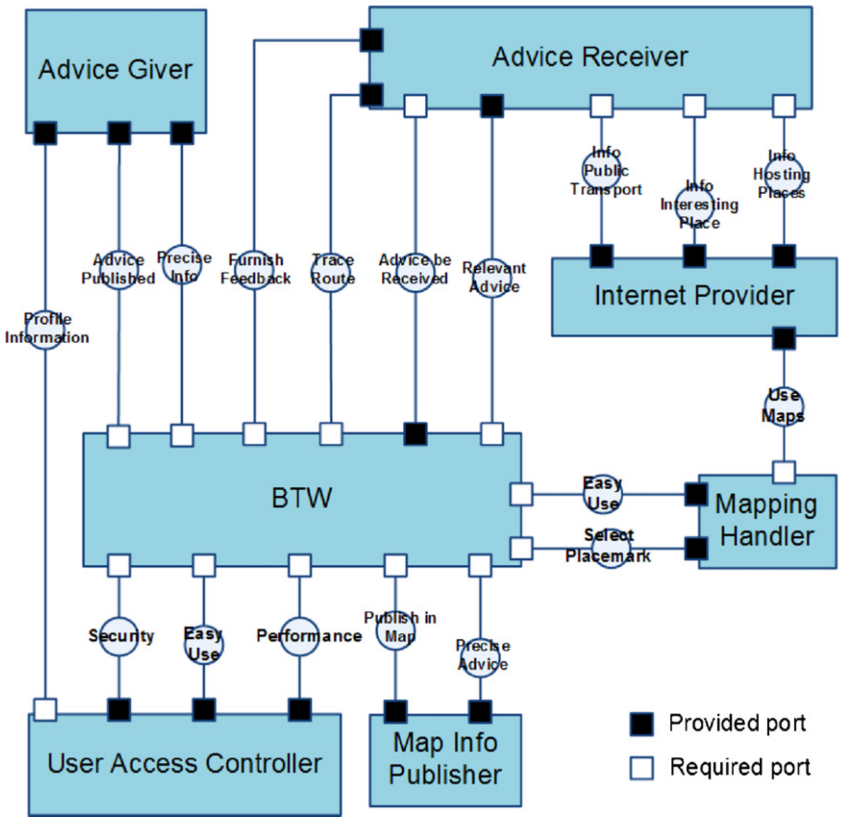


Fig. 6. BTW architectural model produced by the third activity of STREAM.

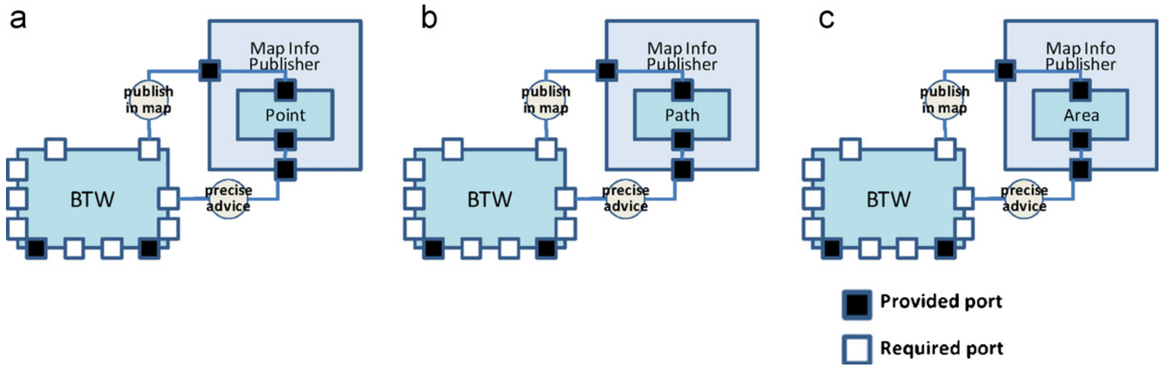


Fig. 7. Architectural solutions for the BTW system.

vice represented by the task (Table 7c, line 6). For example, in the BTW-UFPE project, there is the *Select Placemark* task dependency. Thus, the *Mapping Handler* component has a provided port responsible for offering the service represented by the *Select Placemark* task.

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 (Table 8c, line 9). This type represents what will be returned (provided) by that component. This mapping represents that while a service (task) is performed by a component (actor), an information (resource) is produced and provided through the port. In BTW-UFPE project, there is one case of resource dependency that is *Profile Information*. Thus, a property that returns this resource is assigned to the provided port of *Advice Giver* component.

A softgoal dependency is similar to a goal dependency but its fulfillment cannot be precisely defined. 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 of the connector representing the dependency (Table 9c, line 9). This enumerated type is used to describe the degree of satisfaction of the softgoal (satisfied, partially satisfied, conflict, unknown, partially denied, and denied).

For the BTW example, the *Map Information Publisher* component has a provided port connected to the *Precise Advices* and inside this port there is a property of type enumeration representing the *Precise Advices* softgoal. The same mapping is performed for other softgoal dependencies between actors representing software components.

Therefore, all software actors present in Fig. 5 are mapped to the following components: *BTW*, *Mapping Handler*, *User Access Controller*, *Map Information Publisher*, *Advice Receiver*, *Advice Giver* and *Internet Provider*. Fig. 6 presents the BTW architectural model in Acme generated from the application of the vertical transformation rules.

Observing Fig. 5b, note that inside the *Map Information Publisher* actor, there are three alternative ways (via *path*, *area* or *point*) to achieve the *Information be Published in Map* goal. i* uses means-end relationships to represent alternatives that accomplish a goal. In Acme, the concept of representations is used to describe alternatives. Thus, each alternative of Map Information Publisher actor was mapped to a representation of Map Information Publisher component, since each alternative is modeled as a different representation of the *Map Information Publisher* component (Fig. 7a–c). Hence, there are three architectural design solutions for the BTW software. Each solution has a different internal representation for *Map Information Publisher* component, addressing alternative information to be published in the map (*path*, *area* or *point*). In the final architectural design description, only one of these representations will remain. This choice is performed in the next activity of the STREAM process.

4.3. Select an architectural solution

After generating the full set of possible architectural solutions, the architect needs to decide the most appropriate one for a given requirements specification. Softgoals, present in the requirements models, provide valuable information to assist the software engineer in the decision.

Thus, this third activity receives as input the modular requirements model (Fig. 5) and a set of possible architectural solutions represented by Fig. 6 complemented by Fig. 7. These artifacts are required because the set of architectural solutions are analyzed to check how they satisfy the softgoals captured by the requirements model. In fact, the architectural solution presented in Fig. 6 shows a high level description. It does not show details about elements that are inside the components. These kinds of details are present inside *Map Info Publisher* component to show that this component has three internal representations (Fig. 7).

Note that, in our case study, the architectural model produced encloses three architectural solutions (Fig. 7), since the *Map Info Publisher* component has three potential representations. The choice of which alternative to be used can be based on the contribution of each solution to the satisfaction of the softgoals captured in the requirements model. Observe in the *Map Information Publish* actor (Fig. 5b) that there are three alternatives to fulfill the *Information be published in map* goal that contributes to the satisfaction of *Precise Advices* softgoal. The *Write Information about a point* task contributes very positively (*make/++*) to the satisfaction of the *Precise Advices* softgoal. While *Write Information about a path* task contributes positively (*help/+*) and the *Write Information about an area* task contributes negatively (*hurt/*−) to the achievement of the softgoal. Therefore, according to the strength of the contribution, *make/++* has priority over the other ones. Thus, the *Point* alternative (Fig. 7a) is selected as the most appropriate component for the *Map Info Publisher* component.

The next activity is related to the use of styles or patterns to refine the architecture.

4.4. Refine architecture

Having produced an early architectural design solution, we can now refine it. This activity relies on some commonly used architectural patterns, such as model view control (MVC), layers and client-server (Castro et al., 2003). The components of the architectural model will be manually refined by the architect based on his/her expertise. The choice of the most appropriate pattern should consider many factors, including NFRs.

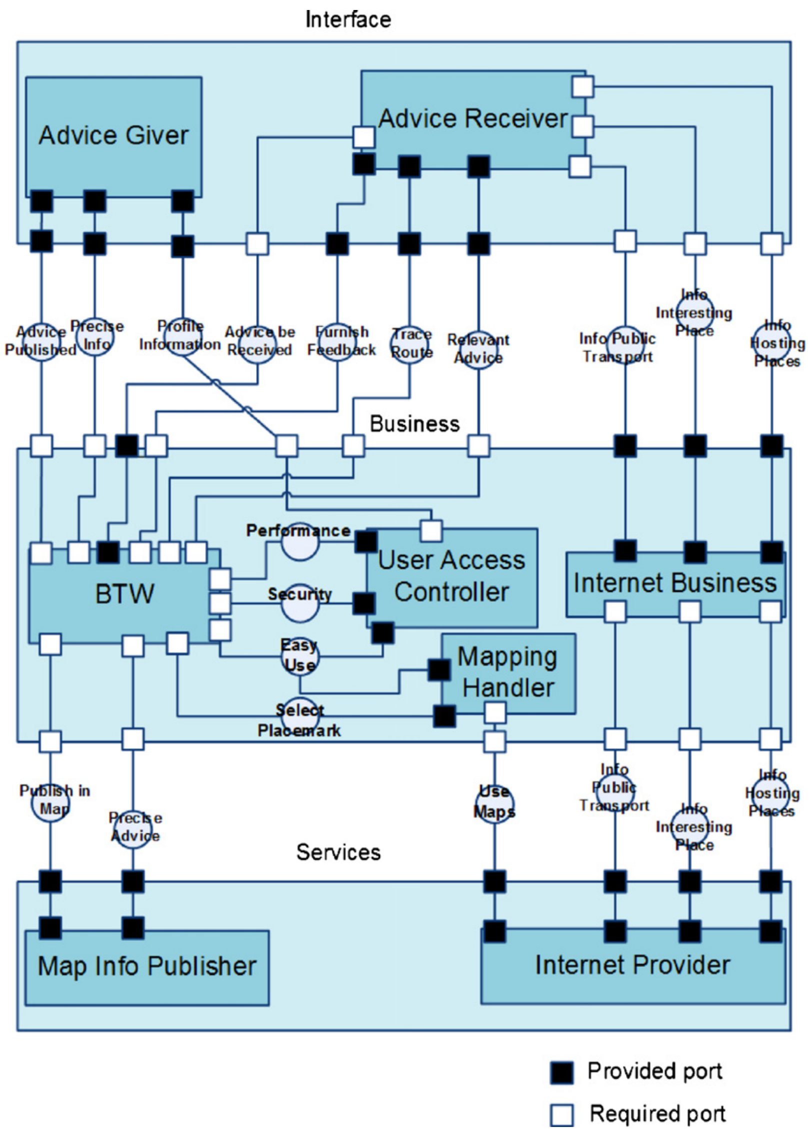


Fig. 8. BTW architecture after applying the Layer architectural pattern.

In our case study we have chosen the Layers pattern, since in the BTW software there are high and low-level services, so that the Layers pattern is a natural candidate. In fact, modularization metrics (see Section 4.5) could be used to check if this choice has improved modularity, one of our concerns. Besides, this pattern contributes positively to *Security* and does not affect the *Usability* NFR (Buchmann et al., 1996), other key quality attributes (together with *Performance*) present in Fig. 5a.

Usually the Layers pattern suggests separating software in the following layers: interface, business and services. Each layer has a specific functionality and upper layers depend on lower layers. In our case study, we identified the three layers previously mentioned, as shown in Fig. 8. The lower layer is *Service*, since it aggregates components that provide services available over the Internet. The middle layer has components related to the *Business* logic. The top layer components interact directly with the users and, for that reason, is called *Interface*. In general, there is no systematic way to apply a pattern. What we have are principles that software architects usually adopt. Therefore, in this work we group these principles into general steps that guide the application of the Layer pattern.

As layers group a set of components, the layer itself is represented as a component. This kind of abstraction (grouping components inside other components) is allowed by Acme. The internal components arrangement forms a sub-architecture that refines the layer. The layer preserves the interface (set of ports) of the components with the components outside the layer. For each port that is connected with a component outside the layer, we will have the same port in the layer. A bind associates the internal port with the external port.

The first step is to analyze the components of the original architecture (e.g., BTW) and compare them with the elements of the pattern (e.g., Layers) observing the similarities of their roles and responsibilities. As commented earlier, the Layers pattern is characterized by layers that depend only on the layer below it. Therefore, we rearranged the components of the early architectural design (Fig. 6) to conform to this constraint. Components at the bottom layer provide services to the components above them, and so on. There is no restriction for components at the same layer. Therefore, components that mutually rely on each other must stay in the same layer. To apply the Layer pattern, we must group the components in each layer according to their roles and responsibilities.

The second step is to analyze the connectors of the original architecture and compare them with the pattern connectors. If they are similar, they need to be associated. In our case study, there was no need to create or remove connectors, since there is no defined role for the connectors of the Layer pattern.

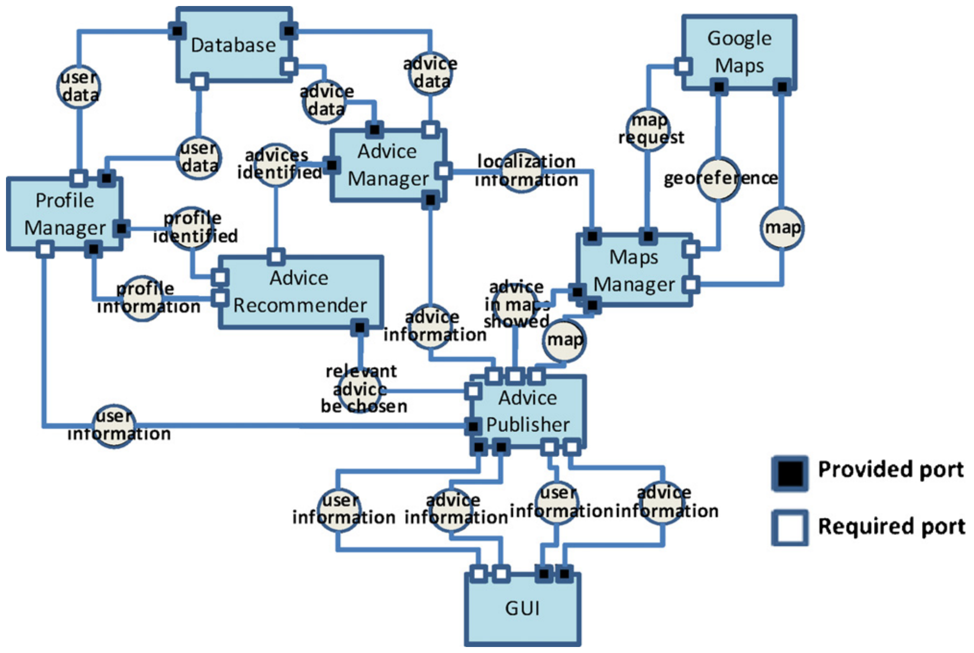


Fig. 9. BTW architecture of the SCORE project.

The third step is to introduce components to adjust the architectural model to the pattern. The strict definition of the Layers pattern requires that a layer must depend only on the layer immediately below it. In the BTW Software architectural model, the *Advice Receiver* component (top layer) relies on the *Internet Provider* component (bottom layer). This connections cross the middle layer towards the bottom layer. Therefore, in order to respect the strict definition of the Layers pattern, a new component, called *Internet Business*, must be introduced in the middle layer to provide internet services to the top layer (see Fig. 8).

Note that the architecture being represented in the modular i* model (Fig. 5(a)) is the system architecture, so that humans and external software that interact with the software-to-be are modeled as actors. The architecture represented in Fig 8 is the software architecture, so that each component was mapped according to the following: (i) human actors are mapped to the Interface Layer (e.g., Advice Giver and Advice Receiver); (ii) external software actors are mapped to architectural components located at the Service Layer (e.g., Internet Provider), and; (iii) actors representing the software-to-be are mapped to architectural components located at the Business Layer (e.g., BTW).

In order to make a comparison between the BTW architecture description generated by the STREAM approach and the one defined by the BTW-UFPE team for the SCORE competition (Pimentel et al., 2010), let us review this latter architecture also described in Acme (Fig. 9).

In this BTW-UFPE team's architectural model, five different components were identified: *Advice Manager*, *Profile Manager*, *Advice Publisher*, *Advices Recommender* and *Maps Manager*. The *Profile Manager* component is responsible for maintaining profiles of system users. The *Advice Manager* is responsible for the maintenance of advices. The *Advice Publisher* is responsible for the publication of advice in accordance with the user profiles using the resources of graphic presentation of maps. The *Advices Recommender* is responsible for reviewing the profiles, advices and recommendations to suggest routes or areas to be visited by travelers. The *Maps Manager* is responsible for maintenance of maps depending on external maps management components.

Note that this architectural solution reflects the real implementation of the BTW software presented at the SCORE contest (Pimentel et al., 2010).

4.5. Evaluating architectural alternatives

An evaluation of the software architectural design while it is still a candidate specification can greatly reduce project risk. The comparison of the candidate architectures is highly dependent on the quality attributes of interest (such as performance, availability, extensibility, security, usability, and modularity), as well as on their associated architectural metrics and process for analyzing them.

In this paper, we have only conducted a partial evaluation of our approach, focused on its main goal, i.e., the improvement of the modularity of requirements specifications and architectural design descriptions. This key non-functional attribute has direct effect on reusability and scalability of models (Sant'Anna et al., 2007a). Thus, we can check if the design decisions taken for the architectural model, such as the one related to the use of the *Layers* pattern, have indeed improved the modularity of the architectural design description. Certainly, an evaluation that encompasses more quality attributes could be performed but it is currently out of scope of this work.

In order to evaluate the architectural design alternatives produced by both the STREAM approach and the BTW-UFPE team, we used a set of modularity metrics proposed by Sant'Anna et al. (2007a,b). We have selected this set of metrics because they were deployed to evaluate architectural design described using an ADL, such as Acme. Table 10 shows the suite of modularity architectural metrics used to evaluate cohesion, concerns diffusion and coupling.

We used the *Lack of Concern-based Cohesion metric* (*LCC*) to measure the cohesion attribute. This metric evaluates how cohesive are the components according to the separation of concerns. In the case of a component with high cohesion, it is expected that it deals with few concerns.

The concern diffusion attribute is evaluated with the metrics of *Concern Diffusion over Architectural Components* (*CDAC*) and *Concern Diffusion over Architectural Interfaces* (*CDAI*). CDAC evaluates how many components contribute to the achievement of a concern.

Table 10
Suite of architectural modularity metrics (Sant'Anna et al., 2007a).

Attribute	Metric	Definition
Component cohesion	Lack of concern-based cohesion (LCC)	It counts the number of concerns addressed by the assessed component.
Concern Diffusion	Concern diffusion over architectural components (CDAC)	It counts the number of architectural components that contribute to the realization of a certain concern.
	Concern diffusion over architectural interfaces (CDAI)	It counts the number of interfaces that contribute to the realization of a certain concern.
Coupling between components	Afferent coupling between components (AC)	It counts the number of components that require a service from the assessed component.
	Efferent coupling between components (EC)	It counts the number of components from which the assessed component requires a service.

In this case, the lower is its value of the metric, the lower is the spreading of concerns. This indicates a high separation of concerns among the components of the architecture. CDAI evaluates how many interfaces contribute to the achievement of a concern. In this case, the lower is its value, the lower the concerns diffusion and, consequently, the greater is the separation of concerns.

The coupling attribute is evaluated using *Afferent Coupling between Components* (AC) and *Efferent Coupling between Components* (EC) metrics. These metrics measure the interaction between the components. In this case, the higher is theirs values, the greater is the dependency degree between the components.

The following steps were used to apply this metrics suite:

1. Define a set of concerns that will drive the evaluation.
2. List all components and interfaces (ports in the Acme notation).
3. Label all components and interfaces with the concerns that will be evaluated (the components and interfaces can present more than one label).
4. Collect metrics.
5. Evaluate the results according to properties of concern diffusion, coupling and cohesion.

In our case study, four concerns were considered: *Recommendation*, *Access Control*, *Mapping Handle* and *Information in Maps*. Note that, to perform the evaluation and comparison between two or more architectural design models, the following assumptions were used:

i. the architectural models should be described in the same ADL;
ii. the interfaces (or ports in Acme) that are used by two or more components can be counted several times (one time by each component that requires it);
iii. the interfaces (or ports) with the same name, but in different components, will be considered as different interfaces.

Three versions of the BTW architectural design were compared according to the modularity properties (*concerns diffusion*, *coupling* and *cohesion*). It is expected that modular architectures have high cohesion, low coupling and low concerns diffusion. The evaluation included the BTW-UFPE team's architectural design (BTW-SCORE), the early architectural solution produced by the STREAM approach (BTW-STREAM) and a refined version of the architectural solution after applying the Layers pattern using the STREAM approach (BTW-STREAM-Layers). Using the modularity metrics, we obtained the results shown in Table 11. For all metrics, the lower the value the better. It is important to remember that the LCC metric measures the lack of cohesion, so that a low value means high cohesion.

The values presented in Table 11 show that the two architectural design alternatives produced by the STREAM approach presented higher cohesion if compared to the BTW-SCORE architectural design. It means that the application of the STREAM approach can improve the cohesion to some degree. The architectural design produced by BTW-STREAM and BTW-STREAM-Layers present the same value of LCC. It happens because many of the concerns that were separated into components are now grouped into one layer in the BTW-STREAM-Layers architecture. The separation of concerns is a property of modularity that is measured by the concerns diffusion metrics. According to the results presented in Table 11, the architecture alternatives produced by the STREAM approach have reduced the concerns diffusion over architectural artifacts (i.e., components and interfaces). The coupling attribute, measured by two metrics, show how the components depend on each other. Higher coupling indicate higher dependency among the architecture components. According to the values measured by the AC and EC metrics, the STREAM approach helped to reduce the coupling among components in the architectural design model. The BTW-STREAM-Layers architecture presented the best results by significantly reducing the coupling among components when compared with the other architectural design alternatives.

In general, the STREAM approach improves the modularity of the architectural design when compared to the BTW-SCORE architectural design. The results for BTW-STREAM and BTW-STREAM-Layers were very close for cohesion and concerns diffusion attributes. However, we probably would select BTW-STREAM-Layers by its expressive gains in the coupling attribute.

5. Discussion

It is worth noting that our approach is specifically aimed at supporting the derivation of Acme architectural design models from i*

Table 11
Results after applying architectural metrics.

Attribute	Metrics	BTW-SCORE	BTW-STREAM	BTW-STREAM-Layers
Cohesion	Lack of concern based cohesion (LCC)	12	9	9
Concern diffusion	Concern diffusion over architectural components (CDAC)	12	8	9
	Concern diffusion over architectural interfaces (CDAI)	20	15	13
Coupling	Afferent coupling between components (AC)	15	11	3
	Efferent coupling between components (EC)	15	9	3

requirements models. Hence, the heuristics proposed in our work correspond to transformation rules oriented to these languages. Furthermore, if different languages are selected, new transformation rules are required. The development of effective heuristics may require a large experience from the software engineering relation to the chosen languages.

The use of metrics, as presented in Section 4.5, requires special attention. Given that the STREAM approach is aimed at improving the reusability and scalability of the models, it is natural that the evaluation provided in this paper was focused on modularity metrics. However, many other quality attributes (and associated metrics) could have been used for evaluating architectural design (Bobrica and Niemela, 2002), such as availability, performance and security. The definition, usage or validity of these other metrics are important issues, but are out of scope of this paper.

Goal oriented approaches have long been proposed for the reasoning and understanding of requirements (van Lamsweerde, 2001). They offer a unified framework in which both functional and non-functional concerns can be integrated. Besides, refinement/abstractions links are precisely defined and provide the basis for various forms of qualitative, quantitative or formal reasoning. In this paper, we address the issue of the generation architectural models considering goal models as source models.

We can also highlight some approaches that produce architectural design considering goal models as source models: i* (Bastos and Castro, 2005), KAOS (van Lamsweerde, 2003) and AOV-graph (Silva et al., 2007). For example, the SIRA approach (Bastos and Castro, 2005) focuses on a systematic way to assist the transition from requirements models in i* to architecture. It describes a software system from the perspective of an organization, as stated by the Tropos methodology (Castro et al., 2002). Both requirements and architectural design models are described in term of the i* language. An organizational architectural style is chosen based on a catalogue of NFRs presented in (Kolp et al., 2006). i* elements, at requirements level, are grouped inside an actor according to their contribution to achieve some responsibilities. Then, an architectural design model is created by considering the similarities between the requirements actors and the architectural actors present in the chosen organizational architectural style. In our STREAM approach, we also use i* goal model as input, but we group i* elements into an actor according to their independence in relation to the application domain and the possibility of that actor to be reused in another domain. The i* modularized model is then mapped to the Acme ADL elements to reach early architectural design solutions. We also choose among alternative architectural design solutions and traditional architectural patterns (Buchmann et al., 1996) based on NFRs. Moreover, we can apply architectural patterns and we use, as target model, a generic architectural description language (Acme).

Another approach that also advocates the transformation of models during the early requirement analysis phase was presented in Bresciani et al. (2002). It proposes an iterative process based on successive transformations to incrementally refine the social environment model of the system-to-be. The produced model is richer than the original model. Our approach, on the other hand, is concerned with applying transformations to an i* model aiming at obtaining a more modular i* model that makes it easier to produce an early architectural design model in Acme, also using model transformations.

Axel Lamsweerde defines a method to produce architectural design models from KAOS requirements models (van Lamsweerde, 2003). In his approach, requirements 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 relies on KAOS modeling language, which consist of a graphical tree and a formal language. In our STREAM approach, we use another goal model as input, the i* model. In fact, we advocate that first we need to modularize the i* models by means of horizontal transformations. Then we use vertical transformations to refine them according to architectural patterns. The mapping from i* models and architectural design models is made easier by the presence of actor and dependency concepts. Although KAOS encloses the concept of agents, it does not support the concept of dependencies among them.

In Silva et al. (2007) a set of mapping rules is proposed between the Aspectual Oriented V-graph (AOV-graph) and the AspectualACME, an ADL based in Acme. Each element (goal/softgoal/task) present in an AOV-graph is mapped to an element of AspectualACME, depending on its position in the graph hierarchy. The information about the source of each element in the AOV-graph is registered in the properties of a component or a port in AspectualACME. These properties make it possible to keep the traceability and propagation of change from AspectualACME to AOV-graph models and vice versa. In our approach, NFRs are considered for the selection among architectural design models. Later this early architectural design model is refined according to an architectural pattern chosen using as criteria the NFRs.

In the context of this work, we are working specifically with i* models to produce architectural models in Acme (Garlan et al., 1997). To the best of our knowledge, there are no studies using i* as requirements models that generates architectural design descriptions in Acme. Moreover, it is well known that goal-oriented requirements specifications present complex representations. The first activity of the STREAM process is a first step to address this issue.

Nonetheless, an experienced architect could define his/her own transformation rules based on the model transformation by example approach (Varro, 2006), for instance. Such approach is used in García-Magariño et al. (2009) to support specific requirements refinements and agents' communication specification.

The use of design patterns is a good way of providing experience and best practices with a view of increasing reusability. Bézivin et al. (2005) focus on the use of design patterns and said that they were working on the construction of a collection of 23 patterns in the context of Model Driven Engineering (MDE). The research covers the design of metamodels, transformations, compositions and other operations on models. In particular, transformation parameters and multiple matching patterns were described using the Atlas Transformation Language (ATL) [47].

In our work, we outlined some transformations. However, they are not yet properly described. We expect in the near future to be able to accurately describe them in ATL. The use of MDE design patterns, as proposed by Bézivin et al. (2005), could be of great assistance in our future work. The idea is to identify any similarity between our transformations and the patterns proposed by Bézivin, trying to adapt them to the context of goal-oriented modeling.

The work on Le Goaer et al. (2008) proposes a reuse-based framework for architecture evolution, which is guided by evolution styles. However, it does not describe how to define an initial architectural design that latter are to be evolved.

The i* modeling language is rich and expressive in describing the system requirements (Yu, 1995). The approaches that use i* modeling language as the starting point of software specification, such as RISD (Grau et al., 2005), Tropos (Castro et al., 2002), and PRIM (Grau et al., 2006), do not support a systematic transition from requirements specifications to architectural design description. Often architects perform architectural design in an ad hoc manner and do not benefit from all the expressiveness and richness offered by the requirements models. Filling this gap between requirements engineering and architectural design activities will allow the i* models to drive subsequent software development phases, relating requirements models to architectural design models, in order to make the developed software systems closer to the stakeholders needs.

6. Conclusions and future works

This paper presented the STREAM approach, that includes a process to generate an architectural model described in Acme, from i* requirements models. The first activity of the process prepares an i* requirements model to balance the responsibilities of a software actor, delegating them to other new software actors. An early set of horizontal rules was defined in Lucena et al. (2009a) to generate these intermediate models (more modular i* models), which are closer to architectural design models.

From these intermediate models, the second activity of the STREAM process derives a set of possible architectural design solutions described in Acme (Garlan et al., 1997). Our vertical transformation rules relate requirements and architectural models, allowing better traceability and propagation change between them. Furthermore, the use of a more general architectural language, such as Acme, led us to propose more generic mapping rules, which, in turn, can serve as a guide to derive architectural design models using other ADLs.

The third activity of our process is related to the selection of an architectural solution using as criteria a given set of NFRs. From these architectural design solutions, described in Acme, it is possible to choose an architectural design solution that better achieves the NFRs present in the i* SR model.

Finally, the fourth activity refines the architecture. It is inspired by architectural patterns proposed in Buchmann et al. (1996) and produces a detailed architectural design solution. This activity uses as input the architectural design solution chosen by the previous activity. In our case study, we showed how a given architectural style (e.g., the Layers pattern) could be used to generate a more refined architectural design model.

We evaluated our approach using a suite of metrics applied to a web recommendation system (BTW) that is an awarded system developed for the SCORE contest at ICSE 2009 (Pimentel et al., 2010). The use of the STREAM approach to this real system has shown that we can produce an architectural design solution with improved separation of concerns. A suite of metrics was used to evaluate the architectural design developed for the contest (BTW-SCORE) against the architectural design produced by the STREAM approach (BTW-STREAM and BTW-STREAM-Layers). The architectural design models obtained using the STREAM process have presented better results, in relation to the modularity property, than the architectural design model produced by the BTW-UFPE team.

We are exploring in more depth the MDE design patterns, as proposed by Bézivin et al. (2005), in order to identify any similarity between the patterns and our transformation rules, which will be formally described in ATL. Therefore, we should adapt the patterns to the context of goal-oriented modeling. We are also investigating how to incorporate our transformation rules, specified using ATL (Jouault and Kurtev, 2005), in the iStarTool (Anon., 2009). This automation will allow us to investigate the scalability of our approach to other real life software systems projects.

Besides, we also need to consider other factors that architects take into account when designing software architecture, such as cost, the involved technology and the familiarity with an already known architecture. These factors are closer to the solution space than the NFRs.

Acknowledgments

This work has been supported by the Brazilian institutions Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), and by the Erasmus Mundus External Cooperation Window - Lot 15 Brazil.

References

Alencar, F., Castro, J., Lucena, M., Santos, E., Silva, C., Araújo, J., Moreira, A., 2010. Towards modular i* models. In: Proc. 25th SAC International Conference—RE Track ,. ACM, New York, pp. 292–297.
Anon., 2009. IStarTool Project: A Model Driven Tool for Modeling i* Models. http://portal.cin.ufpe.br/ler/Projects/IstarTool.aspx.
Bastos, L., Castro, J., 2005. From Requirements to Multi-agent Architecture Using Organisational Concepts, vol. 30. ACM SIGSOFT Software Engineering Notes, pp. 1–7.
Berry, D.M., Kazman, R., Wieringa, R., 2003. Proc. Second International Software Requirements to Architectures Workshop (STRAW'03) at ICSE'03 , Portland, Oregon, USA.
Bézivin, J., Jouault, F., Palies, J., 2005. Towards model transformation design patterns. In: Proc. the 1st European Workshop on Model Transformations (EWMT 2005).
Bobrica, L., Niemela, E., 2002. A survey on software architecture analysis methods. IEEE Transactions on Software Engineering 28 (7), 638–653.
Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J., 2002. Modeling early requirements in tropos: a transformation based approach. Proc. the 2nd International Workshop on Agent-oriented Software Engineering, vol. 2222, Springer. Lecture Notes in Computer Science, 151–168.
Buchmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., 1996. Pattern-oriented Software Architecture: A System of Patterns. John Wiley & Sons, Chichester, UK.
Castro, J., Kramer, J., 2001. Proc. First International Workshop on From Software Requirements to Architectures (STRAW'01) at ICSE'01 , Toronto, Ontario, Canada.
Castro, J., Kolp, M., Mylopoulos, J., 2002. Towards requirements-driven information systems engineering: the Tropos project. Information Systems 27, 365–389.
Castro, J., Silva, C., Mylopoulos, J., 2003. Modeling organizational architectural styles in UML. In: Proc. the 15th International Conference on Advanced Information Systems Engineering (CAISE 2003) , pp. 111–126.
Castro, J., Franch, X., Mylopoulos, J., Yu, E. (Eds.), 2010. Proc. the 4th International i* Workshop, Hammamet, Tunisia, CEUR Workshop Proceedings, vol. 586.
Consortium, O.M.G., 2006. SPEM—Software Process Engineering Metamodel (SPEM) Specification, Version 1.1. http://www.omg.org/docs/formal/05-01-06.pdf.
Czarnecki, K., Helsen, S., 2003. Classification of model transformation approaches. In: Proc. the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture , pp. 1–17.
de Boer, R., van Vliet, H., 2009. On the similarity between requirements and architecture. Journal of Systems and Software 82 (3), 544–550.
Dijkstra, E., 1976. A Discipline of Programming. Prentice-Hall.
Estrada, H., 2008. A service-oriented approach for the i* framework. Ph.D Thesis, Universidad Politécnica de Valencia, 2008.
Estrada, H., Martinez, A., Pastor, O., Mylopoulos, J., 2006. An empirical evaluation of the i* framework in a model-based software generation environment. In: Proc. the 18th International Conference on Advanced Information Systems Engineering (CAISE 2006), LNCS, vol. 4001 ,. Springer-Verlag, Luxemburgo, pp. 513–527.
Franch, X., 2010. Incorporating modules into the i* framework. In: Pernici, B. (Ed.), Proc. the 22nd International Conference on Advanced Information Systems Engineering (CAiSE 2010). Springer, Hammamet, Tunisia, pp. 439–454.
García-Magariño, I., Rougemaille, S., Fuentes-Fernández, R., Migeon, F., Gleizes, M.P., Gómez-Sanz, J.J., 2009. A tool for generating model transformations by-example in multi-agent systems. Proc. the 7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS 2009), vol. 55. Advances in Soft Computing, 70–79.
Garlan, D., Monroe, R., Wile, D., 1997. Acme: an architecture description interchange language. In: Proc. the CASCON 97.
Goulão, M., Abreu, F.B., 2003. Bridging the gap between ACME and UML 2.0 for CBD. In: Proc. Specification and Verification of Component-Based Systems Workshop (SAVCBS'2003) at the ESEC/FSE'2003 , Helsinki, Finland.
Grau, G., Franch, X., 2007. On the adequacy of i* models for representing and analyzing software architectures. In: Proc. ER 2007 Workshops CMLSA, FP-UML, ONISW, QoIS, RIGiM, SeCoGIS ,. Springer, Berlin/Heidelberg/Auckland, New Zealand, pp. 296–305.
Grau, G., Franch, X., Mayol, E., Ayala, C., Cares, C., 2005. RiSD: a methodology for building i* strategic dependency models. In: Proc. the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE 2005) , Taipei, Taiwan.
Grau, G., Franch, X., Avila, S., 2006. J-PRiM: a java tool for a process reengineering i* methodology. In: Proc. the 14th IEEE International Requirements Engineering Conference, RE 2006 ,. IEEE Computer Society Press.
Grünbacher, P., Egyed, A., Medvidovic, N., 2004. Reconciling software requirements and architectures with intermediate models. Software and System Modeling (SoSyM) 3 (3), 235–253.

Hofmeister, C., Nord, R., Soni, D., 2001. Applied Software Architecture. Addison-Wesley.
Jouault, F., Kurtev, I., 2005. Transforming models with ATL. In: Proc. Model Transformations in Practice Workshop (MTIP) at MoDELS Conference , Montego Bay, Jamaica.
Kolp, M., Giorgini, P., Mylopoulos, J., 2006. Multi-agents architectures as organizational structures. Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS) 13, 3–25.
Kotonya, G., Sommerville, I., 1998. Requirements Engineering Processes and Techniques. John Wiley & Sons Inc.
Le Goaer, O., Tamzalit, D., Oussalah, M., Seriai, A.D., 2008. Evolution shelf: reusing evolution expertise within component-based software architectures. In: Proc. the 32nd Annual IEEE International on Computer Software and Applications (COMPSAC 2008) , pp. 311–318.
Lucena, M., Silva, C., Santos, E., Alencar, F., Castro, J., 2009a. Applying transformation rules to improve i* models. In: Proc. the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE 2009) , Boston, USA, pp. 43–48.
Lucena, M., Castro, J., Silva, C., Alencar, F., Santos, E., Pimentel, J.A.H.C., 2009b. A model transformation approach to derive architectural models from goal-oriented requirements models. In: Proc. the OMT Workshop IWSSA, LNCS ,. Springer-Verlag, Berlin/Heidelberg/Vilamoura, Portugal, pp. 370–380.
Nuseibeh, B., 2001. Weaving Together Requirements and Architectures. IEEE Computer 34 (2), 115–117.
Pimentel, J., Borba, C., Xavier, L., 2010. BTW: if you go, my advice to you Project, May. http://jaqueira.cin.ufpe.br/jhcp/docs.
Sant'Anna, C., Figueiredo, E., Garcia, A., Lucena, C.J.P., 2007a. On the modularity of software architectures: a concern-driven measurement framework. In: Proc. First European Conference, ECSA 2007 ,. Springer, Berlin/Heidelberg/Aranjuez, Spain, pp. 207–224.
Sant'Anna, C., Figueiredo, E., Garcia, A., Lucena, C.J.P., 2007b. On the modularity assessment of software architectures: do my architectural concerns count? In: Workshop on Aspects in Architectural Description (AARCH) at AOSD'07 , Vancouver, Canada, pp. 1–4.
SCORE, 2009. The Student Contest on Software Engineering—SCORE 2009, July. http://score.elet.polimi.it/index.html.
Silva, L., Batista, T., Garcia, A., Medeiros, A., Minora, L., 2007. On the symbiosis of aspect-oriented requirements and architectural descriptions. In: LNCS 4765, vol. 75.
Taylor, R.N., Medvidovic, N., Dashofy, I.E., 2009. Software Architecture: Foundations, Theory, and Practice. John Wiley & Sons.
van Lamsweerde, A., 2001. Goal-oriented requirements engineering: a guided tour. In: Proc. the 5th IEEE International Symposium on Requirements Engineering (RE'01) , pp. 249–263.
van Lamsweerde, A., 2003. From system goals to software architecture. In: Formal Methods for Software Architectures. LNCS 2804/2003. Springer, pp. 25–43.
Varro, D., 2006. Model transformation by example. Proc. MoDELS 2006, vol. 4199, Springer. Lecture Notes in Computer Science, 410–424.
Yu, E., 1995. Modelling strategic relationships for process reengineering. Ph.D. thesis, Department of Computer Science, University of Toronto, 1995.
Yu, E., Castro, J., Perini, A., 2008. Strategic actors modeling with i*. In: Tutorial Notes, 16th Intl. Conf. on Requirements Engineering (RE 2008) , IEEE Computer Society, Spain, pp. 1–60.
Yu, E., Giorgini, P., Maiden, N., Mylopoulos, J., 2011. Social Modeling for Requirements Engineering. MIT Press, ISBN 978-0-262-24055-0.

Jaelson Castro is an associate professor at the Universidade Federal de Pernambuco, Brazil, where he leads the Requirements Engineering Laboratory (LER), since 1992. He received his Ph.D. in 1990 from Imperial College, London. His research interests include software engineering, requirements engineering, agent-oriented development, aspect-oriented development, model-driven development, and software product lines. Castro serves on the editorial boards of the International Journal of Agent-Oriented Software Engineering and Requirements Engineering journal. He has served as editor in chief of the Journal of the Brazilian Computer Society (JBCS).

Márcia Lucena is an adjunct professor of the Universidade Federal do Rio Grande do Norte, Brazil, since 1999. She received her Ph.D. in Computer Science from the Universidade Federal de Pernambuco in 2010. Her research interests are requirements engineering, software architecture, and model driven development.

Carla Silva earned her Ph.D. in computer science at the Universidade Federal de Pernambuco, Brazil, in 2007. Currently, she is an adjunct professor at the Universidade Federal da Paraíba, Brazil. Her main topics of research are agent-oriented software engineering, software product lines, aspect-oriented development, and requirements engineering.

Fernanda Alencar joined the faculty of Universidade Federal de Pernambuco, Brazil, in 1988, where she earned her doctoral degree in 1999, and became an adjunct professor in 2000. She spent one year (2008–2009) at Universidad Politécnica de Valencia, Spain, as a postdoctoral fellow. In 2006, she did postdoctoral work at Universidade Nova de Lisboa, Lisbon, Portugal, where she was also on leave in 2008. Her research interests include requirements engineering, agent-oriented development, aspect-oriented development, model-driven development, and organizational modeling.

Emanuel Santos is a Ph.D. student at the Informatics Center of Universidade Federal de Pernambuco, Brazil. His research interests include requirements engineering, goal-oriented software engineering, business process modeling and adaptive systems.

João Pimentel is a Ph.D. student at the Informatics Center of Universidade Federal de Pernambuco, Brazil. His research interests include requirements engineering, goal-oriented software engineering, model-driven development and autonomic computing.