Requirements Eng (2012) 17:259–281
DOI 10.1007/s00766-011-0126-z

Deriving software architectural models from requirements models for adaptive systems: the STREAM-A approach

João Pimentel · Márcia Lucena · Jaelson Castro · Carla Silva · Emanuel Santos · Fernanda Alencar

Received: 16 December 2010 / Accepted: 3 June 2011 / Published online: 23 June 2011

Abstract Some quality attributes are known to have an impact on the overall architecture of a system, so that they are required to be properly handled from the early beginning of the software development. For example, adaptability is a key concern for autonomic and adaptive systems, which brings to them the capability to alter their behavior in response to changes on their surrounding environments. In this paper, we propose a Strategy for Transition between Requirements and Architectural Models for Adaptive systems (STREAM-A). In particular, we use goal models based on the *i** (i-Star) framework to support the design and evolution of systems that require adaptability. To obtain software architectures for such systems, the STREAM-A approach uses model transformations from *i** models to architectural models expressed in Acme. Both the requirements and the architectural model are refined to accomplish the adaptability requirement.

Keywords Requirements engineering · Architectural design · Mapping between requirements model and architectural model · Model-driven engineering · Adaptive systems

J. Pimentel (✉) · J. Castro · E. Santos
CIn, Universidade Federal de Pernambuco—UFPE, Recife, PE 50 740-560, Brazil
e-mail: jhcp@cin.ufpe.br

J. Castro
e-mail: jbc@cin.ufpe.br

E. Santos
e-mail: ebs@cin.ufpe.br

M. Lucena
DIMAP, Universidade Federal do Rio Grande do Norte—UFRN, Natal, RN 59 078-970, Brazil
e-mail: marciaj@dimap.ufrn.br

C. Silva
CCAE—Campus IV, Universidade Federal da Paraíba—UFPB, Rio Tinto, PB58 297-000, Brazil
e-mail: carla@dce.ufpb.br

F. Alencar
CTG, Universidade Federal de Pernambuco—UFPE, Recife, PE 50 670-901, Brazil
e-mail: fernanda.ralencar@ufpe.br

1 Introduction

It is well-known that some kinds of systems present quality attributes—non-functional requirements (NFRs)—that have an impact on the architecture of the system as a whole. These requirements must be elicited, analyzed, and properly handled in the early requirements phase. Otherwise, it would compromise the software architectural design quality. Moreover, some NFRs demand specific approaches and mechanisms to enable their achievement. For instance, it is unlikely that an approach to develop mobile systems is also suitable to develop multi-server scalable systems.

In this paper, we are concerned in developing adaptive systems that require a specific NFR: adaptability [44]. Software adaptability may be defined as the software capability for accommodating environmental changes [44]. Thus, an adaptive system must be able to monitor its environment—identifying the changes on it—and to act on response to that change—adapting itself. Similarly, self-managing systems are those that are capable of adapting as required through self-configuration, self-healing, self-monitoring, self-optimization, and so on—which are also referred to as self-* or autonomic systems. These characteristics become more important nowadays due to

increasing demand for software systems that are expected to be flexible, resilient, robust, recoverable, energy-efficient, and so on [39].

Although several works were proposed for modeling and reasoning on adaptive software—such as [3, 38, 42]—a challenge that still remains is to define a systematic approach to design architectures of adaptive systems satisfying the requirements specifications, as well as supporting the coevolvement of both artifacts. In this paper, we propose the STREAM-A (Strategy for Transition between Requirements and Architectural Models for Adaptive systems) that defines a systematic process to generate architectural design models from requirements models for adaptive systems, based on horizontal and vertical transformations rules. This process is based on the generic STREAM process [35].

Transformation approaches [15] appear as an effective way to generate architectural models from requirements models, recognizing the close relationship between architectural design and requirements specification [8]. Horizontal transformations are those on which the source and target models are in the same level of abstraction (e.g., requirements to requirements), while in vertical transformations, the models are on different abstraction levels (e.g., requirements to architecture) [37]. On the STREAM-A process, horizontal transformations are applied to the requirements models represented as *i** goal-based models, resulting on intermediary requirements models closer to architectural models [36]. Vertical transformations map these intermediary models onto architectural models in Acme [35].

Goal-oriented modeling has been acknowledged in several areas of the software engineering discipline as a suitable way of defining and analyzing organizational expectations and systems requirements [32, 49]. In particular, the *i** framework [47] has become one of the main references for goal-oriented modeling, with a strong community and constantly evolving techniques [10, 48]. We have chosen to use *i** on our approach due to (i) its capability of expressing alternative behaviors; (ii) its mechanisms for refining NFRs; (iii) its support for interactive, iterative analysis over goal models; and (iv) the existing extensions that improve its expressiveness and provide richer reasoning. In particular, we are going to use a context-based extension for the *i** framework that supports adaptability at runtime [2]. For describing the architectural design of adaptive systems, we rely on Acme [21]. This language contemplates all the necessary elements to represent architectures. If required, the architecture could be translated onto architecture description language (ADL) that better suits the development environment to be used, benefiting from the fact that Acme is architecture description interchange language. Moreover, further steps in the development cycle could be proposed to support a model-driven approach, such as [40].

Throughout the STREAM-A process, architectural models will be refined and enriched with additional information—such as context annotations—to guide the identification and analysis of actuators and sensors, essential elements of adaptive systems. Moreover, the adaptive reasoning will be encapsulated in a single component [16] that will be connected to the overall architecture. The adaptation reasoning will be performed in a high level of abstraction and therefore easy to change, without requiring any change on the system's source code.

The remainder of this paper is organized as follows. Section 2 presents the main concepts of the *i** language—used to represent requirements as goal models—and the Acme language, employed to describe architectural models. This section illustrates the usage of *i** models by presenting our running example—the smart home system. Section 3 gives an overview of our approach. Sections 4 and 5 describe the activities of the STREAM-A process, applied to the running example. Section 6 summarizes our work and points out open issues and related works. Finally, yet importantly, Sect. 7 presents the conclusions and future works.

2 Background and running example

This section presents the requirements modeling and architectural description languages used in the proposed process. Along with the *i** notation, we present the running example of this paper.

2.1 Goal modeling with *i**

In goal-oriented approaches [32], the role of requirements engineering (RE) is related to the discovery, the formulation, the analysis, and the agreement of *what* is the problem being solved, *why* the problem must be solved, and *who* is responsible for solving the problem.

As the usage of goals grew in the RE community, there are several techniques where goals are used as a major abstraction, including KAOS [18], NFR Framework [14], *i** [47], V-Graph [50], and Techne [28]. Among these approaches, we chose *i**, which will be briefly presented in this subsection.

*i** defines models to describe both the system and its environment in terms of intentional dependencies among strategic actors [47] (*who*). There are two different diagrams, or views, of an *i** model: the strategic dependency (SD) view presents only the actors and the dependency links among them, while the strategic rationale (SR) view shows the internal details of each actor. Within a SR

diagram, it is defined *why* each dependency exists and *what* is required to fulfill them.

Besides the actor, there are four key elements in *i**: goals, softgoals, tasks, and resources. The goals represent the strategic interests of actors, that is, their intentions, needs, or objectives to fulfill its role within the environment in which they operate. Softgoals also represent the strategic interests of the actors, but in this case, these interests are of subjective nature. They are not measured in concrete terms, but are generally used to describe the actors' desires related to quality attributes of their goals. The tasks represent a way to perform some activity, i.e., they show how to perform some action to obtain the satisfaction of a goal or softgoal. The resources represent data, information, or a physical resource that an actor may provide or receive. Softgoals are usually associated to non-functional requirements, while goals, tasks, and resources are usually associated to system functionalities [48].

There is one kind of dependency related to each one of these four elements. 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.

Figure 1 presents an excerpt of a SD model of a smart home system, which is our running example. A *Tenant*, whom is the user of this system, depends on the *Smart home system* to have the temperature of her house managed, to be safer, to be entertained, and so on. In order to fulfill these dependencies, the system also depends on other actors. For instance, it needs that the Fire department handles emergencies.

In the SR diagram, the actor will be detailed using task decomposition, means-end, and contribution links (Fig. 2). The *means-end* links define which alternative tasks (means) may be performed in order to achieve a given goal (end) (e.g., *Control air ventilator* is a possible means to achieve the goal *Temperature be managed*). The *task decomposition* links describe what should be done to perform a certain task (e.g., the task *Control air ventilator* is decomposed onto the tasks *Turn on air ventilator*, and *Turn off air ventilator*). Finally, the contribution links suggest how a task can contribute (positively or negatively) to

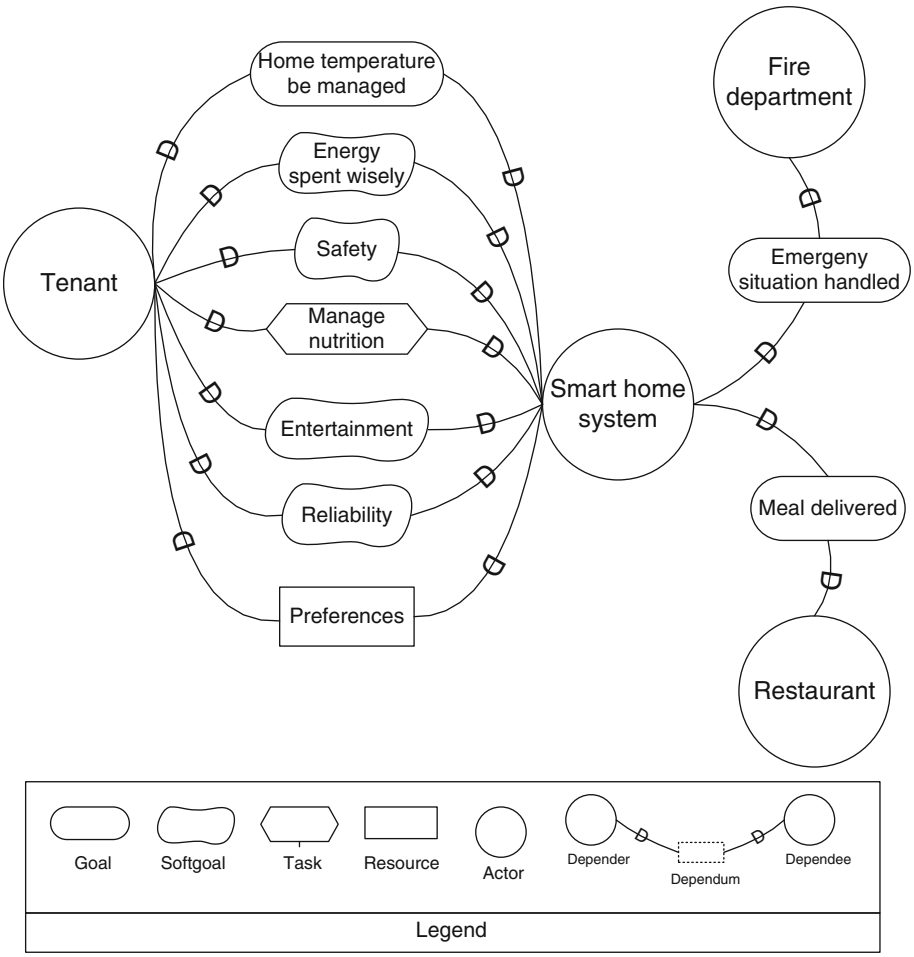


Fig. 1 Strategic dependency (SD) model of a smart home system

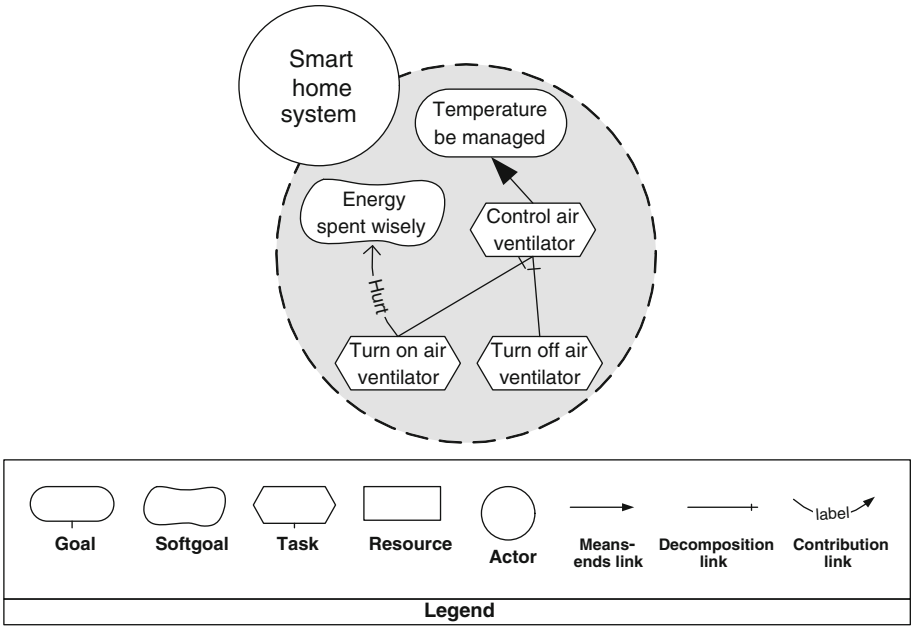


Fig. 2 Excerpt of the smart home system actors' internal elements

satisfy a softgoal (e.g., the task *Turn on air ventilator* contributes negatively to the softgoal *Energy spent wisely*). These contributions allow the selection of alternative tasks driven by the satisfaction of softgoals, which includes non-functional requirements.

Figure 3 presents the complete refinement of the *Smart home system* actor. For the sake of simplicity, the dependencies are omitted, since in the STREAM-A process, only the system itself (software and hardware) will be translated into architecture. However, it is worth noting that the system is mainly a way of satisfying the goals of human actors, such as the *Tenant* actor that will interact with the system.

The highlighted elements in Fig. 3 will be later moved during requirements refactoring, as it is going to be explained in Sect. 4.1.

2.2 Acme architecture models

There is a variety of architectural description languages (ADLs), each one with its own set of tools and techniques. Acme [21] was proposed with the primary goal of providing an interchange format to be used by tools and environments for architectural development. Note that if necessary, it is possible to map Acme diagrams onto other languages such as UML [23].

According to [45], the fundamental elements when describing instances of architectural designs include components, connectors, interfaces, configurations, and rationale. Acme supports each of these concepts, also adding the notion of ports, roles, properties, and representations.

Acme components represent computational units of a system. Connectors represent and mediate interactions between components. Ports correspond to external interfaces of components. Roles represent external interfaces of connectors. Ports and roles (interface) are points of interaction, respectively, between components and connectors. Systems (configurations) are collections of components, connectors, and a description of the topology of the components and connectors. Systems are captured via graphs whose nodes represent components and connector and whose edges represent their interconnectivity. Properties are annotations that define additional information about elements (components, connectors, ports, roles, or systems). Representations allow a component, connector, port, or role to describe its design in detail by specifying a sub-architecture that refines the parent element. Different representations of an element are considered alternative representations of that element. Properties and representations could be associated to the rationale of the architecture, i.e., information that explains why particular architectural decisions were made and what is the purpose of the elements [45].

Acme has both a graphical and a textual language. Figure 4 exemplifies its notation through a client–server example. The client and the server are components, each one with a single port. The connector RPC has two roles: caller and callee. The ports are attached to the connector roles, defining the connection between these two components.

In the following section, we describe our process to systematically transform a requirements model in *i** into

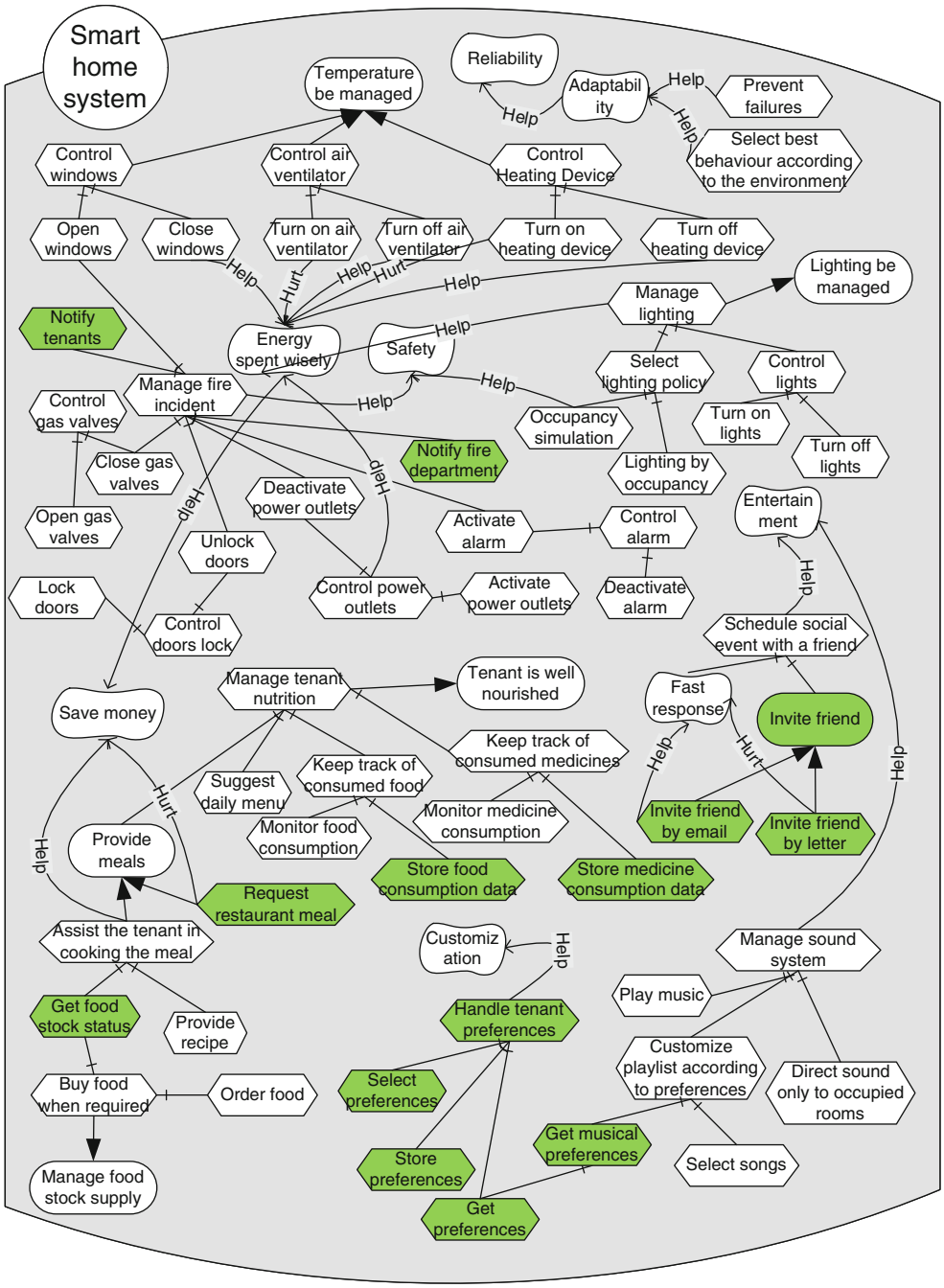


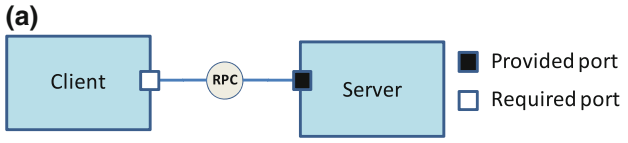
Fig. 3 Strategic rationale (SR) diagram of the smart home system actor

architecture in Acme, aiming at the development of adaptive systems.

3 STREAM-A overview

We defined a process specifically designed for the context of adaptive systems. The goal is to generate architectural models from requirements models, with an incremental and models transformation-based approach. The requirements original model is enriched with the information required to perform the reasoning related to adaptability, and then architecture is derived.

The six activities of this process are depicted in Fig. 5. The first three are related to requirements engineering: *Requirements Refactoring*, *Context Annotation and Analysis* and *Identification of Sensors and Monitors*. The last three are architecture-related: *Generate Architectural*



(b)

```
System example = {
    Component client = {
        Port send-request = {}
};
    Component Server = {
        Port receive-request = {}
};
Connector rpc = {
    Role caller = {}
    Role callee = {}
};
Attachments {
    client.send-request to rpc.caller;
    Server.receive-request to rpc.callee;
};
```

Fig. 4 Client and server components to exemplify the graphical and the textual notation of Acme

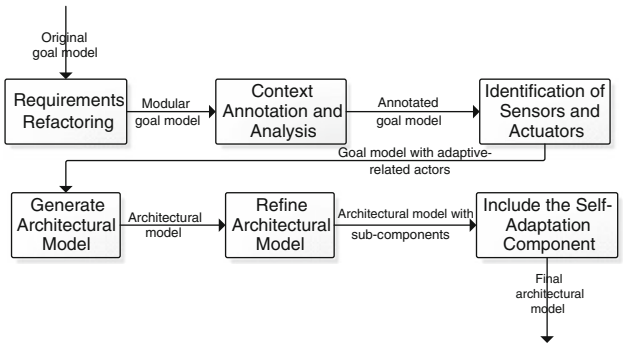


Fig. 5 Overview of the STREAM-A process

Model, *Refine Architectural Model*, and *Include the Self-adaptation Component*. The flow presented in Fig. 5 is just a guidance, as in real-world projects, an iterative approach is more likely to occur.

The *Requirements Refactoring* concerns the modularization of the system. By defining the main elements of the system in this abstraction level, the system components of the resulting architecture are more likely to present a coherent modularization, using the separation of concerns criteria. Systems with good modularization are easier to understand and maintain [11]. To support the refactoring of goal models, a set of transformation rules is defined.

The second activity, *Context Annotation and Analysis*, is based on [2] and defines the environmental contexts that may influence the goals achievement and task executions. The definition of these contexts is crucial for the proper specification of an adaptive behavior. These contexts are analyzed to provide the actual data entities that need to be monitored in order to define the context.

This information will be used in the *Identification of Sensors and Actuators* activity to discover the context sensors that the system will need. During this activity, the context actuators will also be identified, based on the tasks of the goal model.

These last two activities are the most effortful ones of this process, given that they require further elicitation to be performed in order to identify the influencing context and the possible kinds of devices to be used.

At this point, the goal model has sufficient information to start the architecture derivation. The *Generate Architectural Model* is based on a set of transformation rules to map the requirements model onto components and connectors of an initial architectural model.

In the *Refine Architectural Model* activity, the architectural model will be detailed defining the sub-components of the components related to adaptivity.

Finally, in the *Include the Self-adaptation Component*, the architecture will be linked to a component that will perform the reasoning related to adaptivity [16]. This component allows a high-level reasoning, preventing the need of hard coding the adaptation handling.

These six activities will be further detailed in the following sections.

4 STREAM-A—requirements enhancement

The following subsections describe the activities focusing on requirements of the STREAM-A process: *Requirements Refactoring*, *Context Annotation and Analysis*, and *Identification of Sensors and Actuators*.

4.1 Requirements refactoring

*i** models are often overloaded with information that captures features of both the system organizational environment and the software system itself. Its rich diversity of concepts, aligned with misuse of the decomposition mechanisms allowed by the *i** language, can produce models unnecessarily hard to read, understand, maintain, and reuse.

Estrada [20] performed an evaluation of the *i** modeling language regarding modularity management, among other aspects. Modularity of a modeling language measures the degree to which it offers well-defined building blocks for creating a model. The building blocks 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 have to be propagated to other parts. 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 used is not suitable to produce models that are easy to evolve and reuse. Current methods for *i** modeling represent the rationale of an actor in a monolithic way. Sometimes, several refinements are mixed together, making it hard to visualize the boundaries of subgraphs related to specific domains. This poor modularity compromises the management of the complexity of the models, an important prerequisite for the adoption of *i** in industrial settings [20].

The aim of this activity is to improve modularity of the expanded/refined software actor. It allows delegation of different issues of a problem, initially concentrated into a single actor, to new actors. Thus, it is possible to deal with each of them separately, following the separation of concerns principle [19].

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 subactors. We claim that some preprocessing can greatly improve the modularity of *i** models. Thus, this activity relies on using a decomposition criterion based on the separation and modularization of elements or concerns that are not strongly related to the application domain. The domain-independent elements are defined according to the application nature. For example, in a smart home system, the elements related with data persistency or data transmission could be modularized. The decomposition of the main software actor into smaller actors has the objective of modularizing *i** models by delegating responsibilities of the software actor to other (new) software actors that are dedicated to a particular concern. In this paper, we are considering concerns as cohesive groups of domain-independent elements. We have applied heuristics to select the elements that could be in a module [36]. However, defining good criteria is a critical issue that relies on human judgment; thus, other heuristics could be applied if necessary.

For instance, in the smart home system model—whose application domain is home automation—we can identify some elements that are not fully related to the smart home domain. These elements are highlighted in Fig. 3: Notify tenants, Notify fire department, Invite friend, Invite friend by email, Invite friend by letter, and Request restaurant meal, related to communication; Handle tenant preferences, Select preferences, Store preferences, and Get musical preferences, related to the preferences management, and; Store food consumption data, Store medicine consumption data, Get food stock status, and Get preferences, related to data storage.

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

After the identification of the removable elements, they will be transferred to other actors. This is not a trivial step, since the *i** language allows for a high interconnectivity among its elements. In order to assist this step, we propose four model transformation rules. Table 1 presents these rules in terms of their preconditions and effects. A generic example of the application of these rules is presented in Figs. 6, 7, 8, 9, reflecting the terminology used in Table 1. These transformations are improvements on previous rules proposed by [34], contemplating some cases that were missing. In Figs. 6, 7, 8, 9, the top part presents a model before the transformation, while the bottom part presents the same model after the application of the respective transformation rule.

Regarding the taxonomy of model transformations [37], these transformation rules are endogenous, i.e., rephrasing transformations. This is the case since both the source and the target language are the same (*i**). In particular, they are refactoring transformation rules, since they change the model structure to improve its modularity without changing its semantics. Additionally, they are horizontal transformations, since both source and target models present the same abstraction level.

The concept behind these transformation rules is that, in *i**, dependencies are the only relationship allowed for elements of different actors. Therefore, the definition of these rules consists in defining dependencies that preserve the meaning of the original relationships.

The horizontal transformation rule 1 (HTR1) is a transformation rule that simply moves a selected subgraph from the original actor *A* to another actor *A*' (Fig. 6). In doing so, the original relationships among the elements are preserved. Unless the subgraph to be moved is an independent one, this rule will generate a syntactically incorrect model, which will be later corrected. The other three transformation rules will be used to fix these errors. These corrective rules could be, actually, part of the first rule. However, in this paper, we define them separately for the sake of clarity and simplicity.

The horizontal transformation rule 2 (HTR2) considers the situation in which there is a means-end link between elements of different actors (Fig. 7). Since the means-end link has an implicit OR meaning, to preserve the original model semantics, the linked task is replicated inside the

Table 1 Horizontal transformation rules

Horizontal transformation rule	Pre-conditions	Effects
HTR1—Move subgraph between actors	A subgraph *G* is selected to be moved to an actor *A*'; All elements of *G* are within the boundary of a single actor *A*	All elements of *G* are moved to the actor *A*'; For each link between an element of *G* and an element of *A*: If the link is a means-end link, apply HTR2; If the link is a contribution link, apply HTR3; If the link is a task decomposition link, apply HTR4.
HTR2—Move a means-end link crossing the actor's boundary	An element *e* of an actor A has a means-end link *l* to a goal *g* that is inside another actor A'	The element *e* is copied from the actor *A* to the actor *A*', creating a copy element *e*'; The source of the means-end link *l* is moved from *e* to *e*'; The element *e* is replicated as the dependum of a dependency relationship from the copy element *e'* to the original element *e*
HTR3—Move a contribution link crossing the actor's boundary	An element *e* of an actor *A* has a contribution link *c* to a softgoal *s* that is inside another actor *A*'	The softgoal *s* is copied from the actor A' to the actor *A*, creating a copy *s*'; The target of the contribution link *c* is moved from *s* to *s*'; The softgoal *s* is replicated as the dependum of a dependency relationship from the original softgoal *s* to its copy *s*'
HTR4—Move a task decomposition link crossing the actor's boundary	An element *e* of an actor *A* is a decomposition of a task *t* that is inside another actor *A'*	The decomposition link is removed; The element *e* is replicated as the dependum of a dependency relationship from the task *t* to the element *e*

actor that has the goal (actor *A*') and a dependency of the same type is generated linking the replicated task to the original task. Therefore, even if the task is shown inside the actor *A*' (Fig. 7), it will actually be performed (delegated) by the actor *A*.

The third transformation rule (HTR3) handles contribution links between elements of different actors (Fig. 8). Here, the target softgoal is required to be present in both actors. Otherwise, other contributions to the same softgoal (e.g., *hurt* contribution from *Task 6* to *Softgoal 1*) would be affected creating new contribution links between elements of different actors, which this transformation intends to prevent. Therefore, the softgoal will be replicated inside the other actor, the elements of each actor will have their contribution links to that softgoal and a dependency link between the original and the replicated softgoals will be created.

Finally, the horizontal transformation rule 4 (HTR4) defines how to handle task decomposition links between elements of different actors (Fig. 9). In these cases, the decomposition link will be replaced by a dependency link, from the father element to the child one. In the example, this dependency means that to perform Task 5, Task 3 also have to be performed (by Actor Z).

In the smart home system, after applying the horizontal rules in the elements highlighted in Fig. 3, we obtain the result shown in Fig. 10. For the moment, ignore the small (yellow) rectangles and the highlighting of some elements present in Fig. 10, as they are the result of activities to be explained below.

4.2 Context annotation and analysis

The information about context, i.e., the environment on which the system is inserted, is crucial for adaptive systems. It is this information that will allow the system to identify *when* it needs to change its behavior and *what* change is needed. In the words of Cheng et al. [39] "Whenever the system's context changes the system has to decide whether it needs to adapt".

Thus, in this activity, we are concerned with defining the contextual information that will enable the system to behave adaptively. It comprises the enrichment of the requirements model with contextual annotations and the identification of the data that the system will have to monitor. For this, we rely on the goal-based framework for contextual requirements [2]. It allows the insertion of contextual conditions—context annotations—in six different points of a goal model. However, due to a difference of the *i** version used in [2] and the one used in our work, we are considering only five of them, summarized in Table 2. These points are root goal, means-end link, actor dependency, contribution to softgoal, and decomposition link.

Root goal: a context annotation in a root goal means that this goal is supposed to be achieved if and only if the context holds. It can be considered an activation condition. Example: The *Temperature be managed* goal is active only when there is someone at home (C1).

Means-end: a context annotation in a means-end link denotes that a task is actually a means for that particular goal if and only if the context holds. Example: The *Assist*

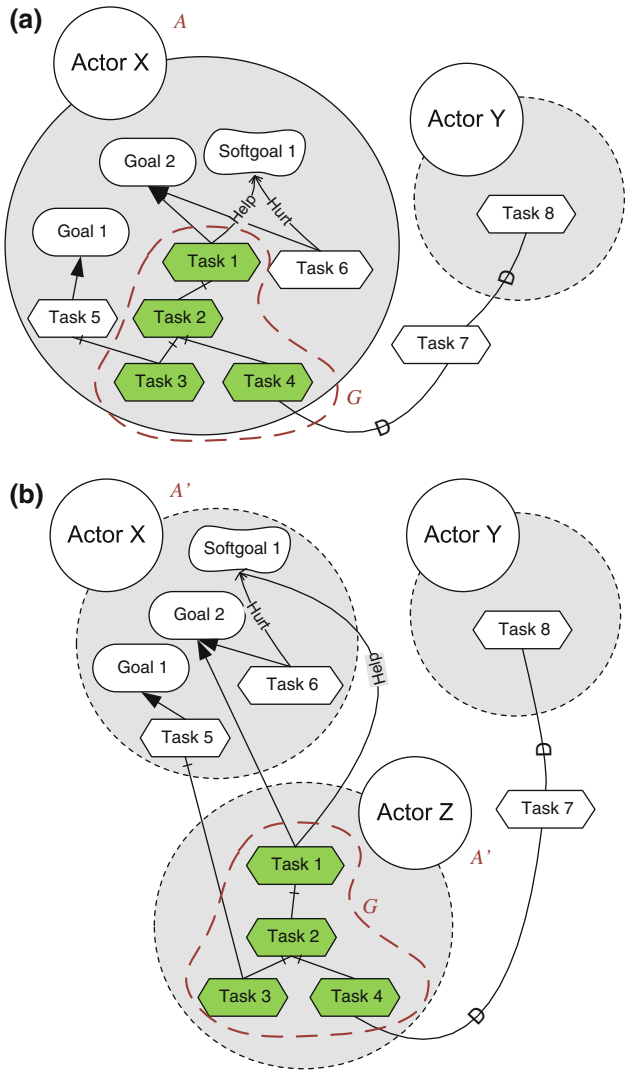


Fig. 6 Example of HTR1—Move subgraph between actors. **a** original model. **b** Result after applying the transformation rule

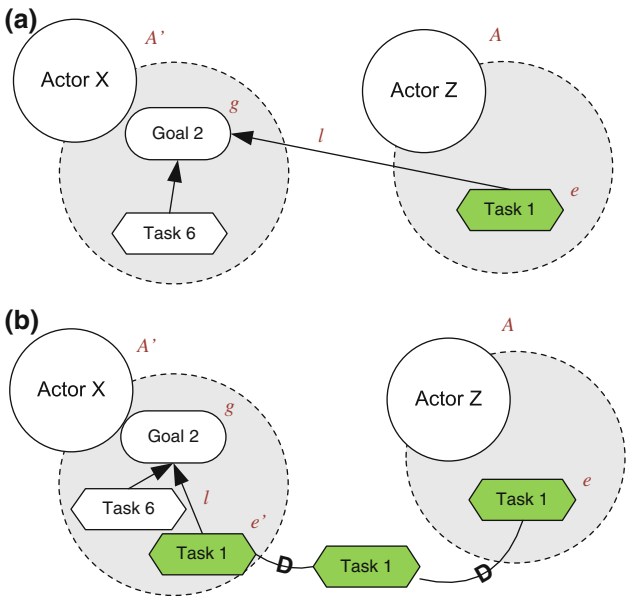


Fig. 7 Example of HTR2—Move a means-end link crossing the actor's boundary. **a** original model. **b** Result after applying the transformation rule

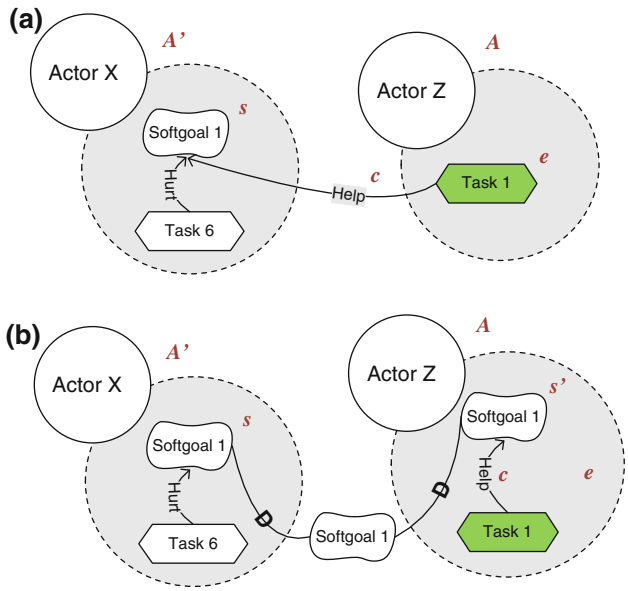


Fig. 8 Example of HTR3—Move a contribution link crossing the actor's boundary. **a** original model. **b** Result after applying the transformation rule

the tenant in cooking the meal is only a means to achieve the *Provide meals* goal if there is enough food on stock (C2).

Actors dependency: a context annotation in a dependency link means that the dependee is capable of fulfilling this dependency if and only if the context holds. Example: The dependency between the *Request restaurant meal* task of the *Smart Home System* actor and the *Communication* actor can be fulfilled only if an Internet connection is available and active at home (C3).

Contribution to softgoal: a context annotation in a contribution link means that this contribution actually exists if and only if the context holds. Example: the *Occupancy simulation* task helps to satisfice the *Safety* softgoal only if there is no one at home (C4).

Decomposition: a context annotation in a decomposition link means that the sub-element is, actually, a sub-element of the decomposed task, only if the context holds. Example: The *Provide meals* goal is part of the *Manage tenant nutrition* task if the tenant is going to eat at home (C5).

In Fig. 10, the context annotations of our running example are presented as (yellow) small rectangles. The description of these annotations is presented in Table 3.

After defining all the contexts that will influence the requirements, they can be analyzed to identify what is

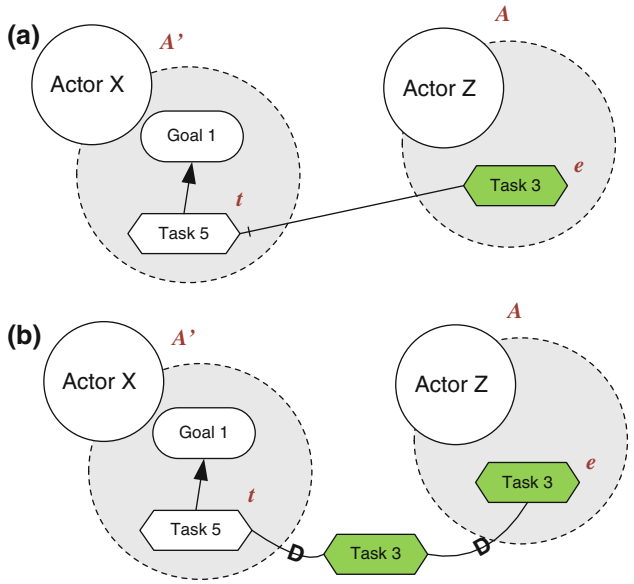


Fig. 9 Example of HTR4—Move a task decomposition link crossing the actor's boundary. **a** original model. **b** Result after applying the transformation rule

the corresponding data in the environment that needs to be monitored by the system. This analysis is also performed as proposed in [2], on which a context model with a tree-like structure is employed. The root of this model is the context, and statements and facts are its nodes. The facts are predicates that can be verified in a context, while the statements cannot be verified in a context. To obtain a verifiable context, all statements are refined into facts and sub-statements, until there are only facts left. With this model, it is possible to identify the variable ways of how the sets of facts can assess a context. Then, the context models are analyzed to eliminate inconsistent statements and define the activation precedence of the facts. Other analysis to trim the variability space can be performed before deriving a preliminary conceptual model with the data necessary to verify the facts. More details about this process are presented at [2]. In our smart home system, the context entities on this conceptual model are the following: luminance, temperature, fire, gas leak, door locks, windows, room occupation, stock supply, Internet connection, power outlets, and alarm device.

It is important to note that the monitoring required to assess the context may have a significative impact on the system under development. For instance, while assessing the temperature might require only a simple and non-expensive sensor, assessing the stock supply might require several sensors and a system to coordinate them. Thus, the impact of monitoring the context data must also be considered when defining the context annotations.

4.3 Identification of sensors and actuators

From the information obtained throughout the previous activity, we are now able to identify the sensors and actuators of the system. Using the definitions of [16], a (context) sensor is "any system providing up-to-date information about the context where the system is running", while a (context) actuator is "any actuator in the environment which can receive commands from the system to act on the environment context", that is , a context sensor monitors an environment and a context actuator performs a change on an environment.

The first step is to create a new actor, named *Monitor*, which will encapsulate all the monitoring tasks. The *Monitor* actor will be linked to the main actor of the system through a goal dependency, on which the dependum is *Environment monitored*. In the main actor, this dependency can be linked directly to the adaptability softgoal or one of its refinements. In our running example, we selected to attach it on a new task, named *Get environment data* (Fig. 11). This new task is a decomposition of the *Select best behavior according to the environment* task.

Then, for each one of the context entities, a new goal will be created, expressing the need to monitor these entities. These goals have the form of *Monitor* [*context entity*], see Fig. 12. Now, we are able to refine these goals and define how to achieve them. For instance, the *Monitor room occupation* goal can be achieved by using a presence detector based on passive infrared sensor or based on a ultrasound one (Fig. 12). Each kind of detector has different impacts to the satisfaction of non-functional requirements, such as *Low Cost* and *Precision*. These impacts—represented as contribution links in *i** models—will guide the selection of the best alternative in the detailed design.

This kind of devices selection is a detailed design decision. However, aligning this selection with non-functional requirements and stakeholders priority for these NFRS allows a more systematic and justified decision. This decision is performed in the *Refine Architectural Model* activity (see Sect. 5.2).

In the smart home system SR model, there are already some tasks related to monitoring: *Monitor food consumption* and *Monitor medicine consumption* (Fig. 10). However, these tasks are functionalities of the system, they are not related to adaptivity itself—for instance, the data about food consumption will be used to manage the patient nutrition; changes on these data will not trigger any adaptation. Therefore, they will not be moved to the *Monitor* actor.

Besides defining the contexts sensors, we also need to define their actuators. This step will be accomplished by moving the tasks that cause a change in the context data to

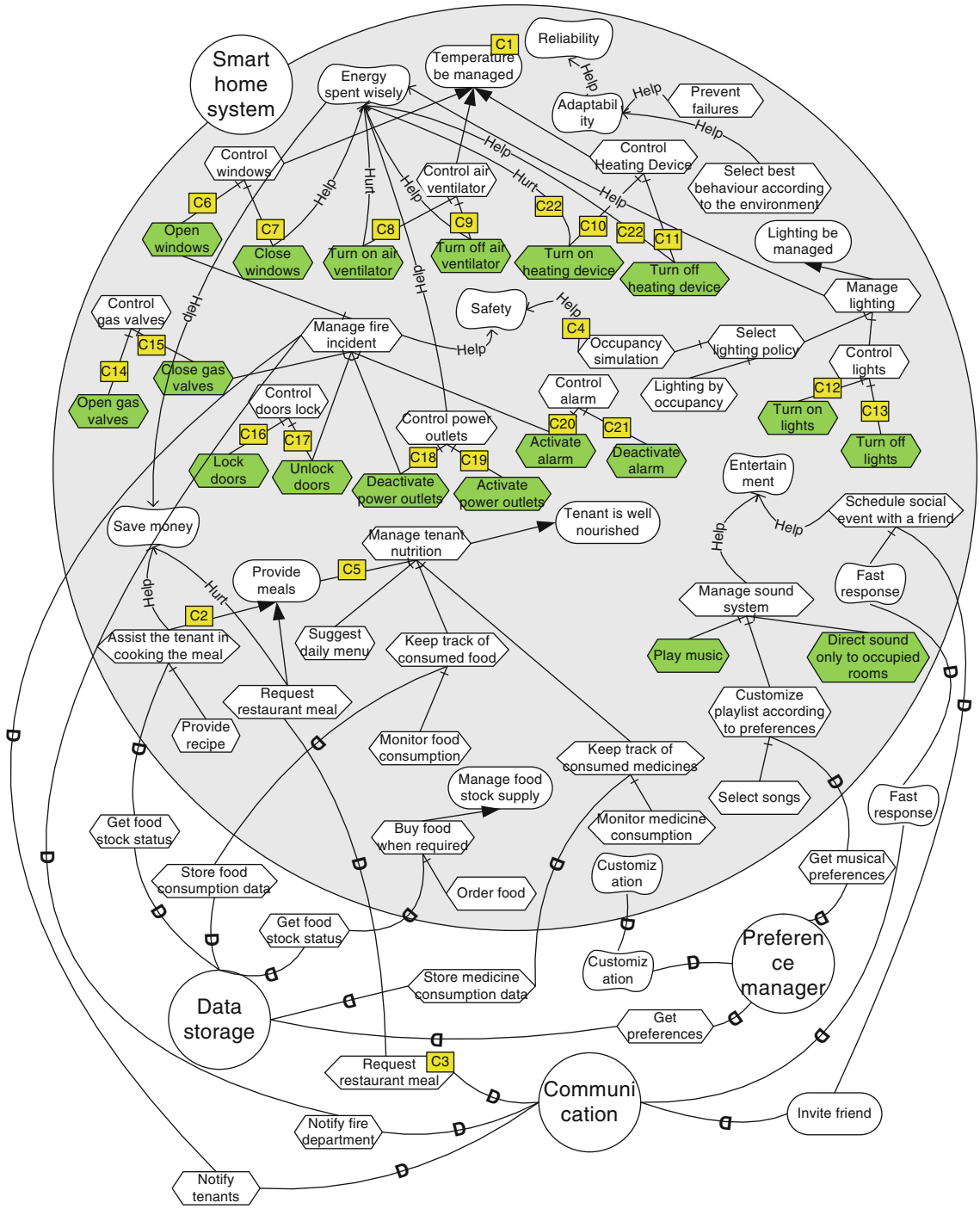


Fig. 10 Refactored goal model with context annotations

a new *Actuator* actor. For instance, the *Turn lights on* task changes the *Luminance* entity of a room, from low luminance to high luminance. In Fig. 10, all tasks that will be moved to the *Actuator* actor are highlighted.

After the identification of such tasks, the same transformation rules, defined in the STREAM-A approach and previously defined (Table 1), can be used to move the highlighted tasks to the *Actuator* actor. The resulting model of our running example is shown in Fig. 13.

Throughout the requirements enhancement phase, we performed the refactoring of the initial *i** model, the context annotations, and the context entities to be monitored

Table 2 Points on which it is possible to insert context annotations on a goal model and their meaning

Activation point	Semantics
Root goal	The annotated goal is supposed to be achieved iff the context holds
Means-end	In the annotated means-end link, the task is actually a means for that particular goal iff the context holds
Actors dependency	In the annotated dependency link, the dependee is capable of fulfilling this dependency iff the context holds
Contribution to softgoals	In the annotated contribution link, the contribution actually exists iff the context holds
Decomposition	In the annotated decomposition link, the sub-element is actually a sub-element of the decomposed task iff the context holds

Table 3 Context annotations on the smart home system

Context	Description
C1	There is someone at the smart home
C2	The food in the house's stock is enough to cook the meal
C3	There is an Internet connection available and active at the smart home
C4	There is no one at the smart home
C5	The tenant is going to eat at the smart home
C6	The temperature at the room is hotter than what would be pleasant for the people within it, the temperature outside is colder than the temperature inside the smart home and, the windows are closed
C7	The temperature at the room is colder than what would be pleasant for the people within it, the temperature outside is colder than the temperature inside the smart home, the smart home is not on fire, and the windows are open
C8	The temperature at the room is hotter than what would be pleasant for the people within it and the air ventilator is off
C9	The temperature at the room is colder than what would be pleasant for the people within it and the air ventilator is on
C10	The temperature at the room is colder than what would be pleasant for the people within it and the heating device is off
C11	The temperature at the room is hotter than what would be pleasant for the people within it and the heating device is on
C12	There is someone at the room or close to it, the room is dark, and the light is off
C13	There is no one at the room or close to it, and the light is on
C14	There is someone at the smart home, there is no gas leaks, the smart home is not on fire, and the gas valves are closed
C15	The gas valves are open
C16	The smart home is not on fire and the door is unlocked
C17	The door is locked
C18	The power outlet is on and there is no vital equipment attached to it
C19	The power outlet is off, the smart home is not on fire, and there is no gas leak detected
C20	The alarm is off
C21	The alarm is on
C22	The heating device is electricity-based

were defined. Moreover, the *Monitor* actor was created and refined and, finally, the *Actuator* actor was defined with tasks from the main actor. These activities transformed the requirements model in *i** into an intermediate (more modular) *i** model closer to an early architectural design. This intermediate model is the input for the architecture derivation phase. In this phase, all the created actors, the main actor, and the dependencies among them will be mapped to architectural elements through vertical transformation rules. These transformations and further architectural refinements are presented in the next section.

5 STREAM—architecture derivation

In this section, each STREAM-A activity that directly deals with architectural models in Acme will be described.

5.1 Generate architectural model

In this step, transformation rules will be used to translate the *i** requirements model onto early architecture model in Acme. Since these transformations have different source and target languages, they are exogenous or translation

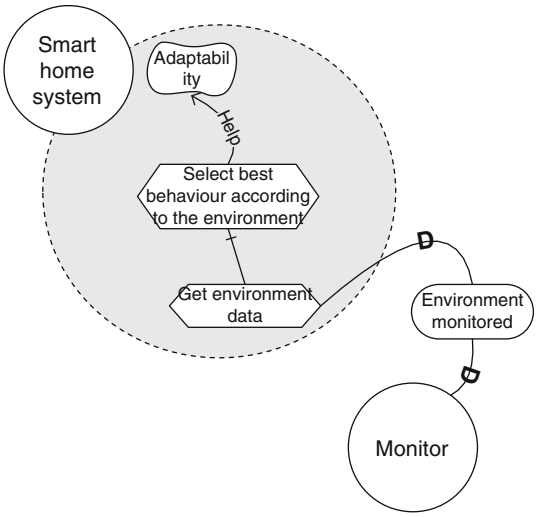


Fig. 11 Excerpt of the goal model depicting the linkage between the main actor (*smart home system*) and the new *Monitor* actor

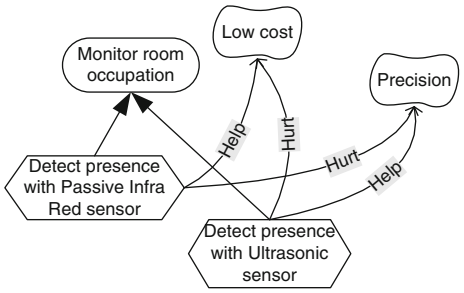


Fig. 12 Excerpt of a refinement of the *Monitor room occupation* goal inside the *Monitor* actor

transformation. They are also vertical transformations, since the source and target models have different levels of abstraction.

In summary, these transformations define the mapping from *i** actors to Acme components and from *i** dependencies to Acme connectors and ports. 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 [45]. An actor in *i** is an active entity that carries out actions to achieve goals by exercising its knowhow [47]. The actor representing the software establishes a correspondence with modules or components [24]. In addition, an actor may have as many interaction points as needed. Hence, an actor in *i** can be represented in terms of a component in Acme.

Our approach deals with the architectural models related with the structural views (e.g., component-and-connector view [4]). It is important to highlight that architectural views based on behaviors and technological issues are not supported yet.

Thus, the first vertical transformation rule is a straightforward one that maps *i** actors onto Acme components. Figure 14a shows an actor in *i** and Fig. 14b shows that actor translated to an Acme component. In Fig. 14c, its textual description is presented. Further details of this component will be added later during the mapping of *i** dependencies.

In *i**, a dependency describes an agreement between two actors playing the roles of depender and dependee, respectively [13]. On the other hand, connectors are architectural building blocks that regulate interactions among components [45]. In Acme, connectors mediate the communication and coordination activities among components. Thus, we can represent a dependency as an Acme connector (see Fig. 15). Interfaces are points of access among components and connectors. In *i**, there is no such thing as a port. 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, depending on the direction of the dependency. Hence, the roles of depender and dependee are mapped to roles that are comprised by the connector (Fig. 15c lines 12 and 13). Thus, we can distinguish between required ports mapped from the depender actor (Fig. 15c lines 2–4) and provided ports mapped from the dependee actor (Fig. 15c lines 7–9). These ports indicate the direction of communication between the depender and dependee components. While in *i** a depender actor depends on a dependee actor to accomplish a type of dependency, in Acme, a component requires that another component carries out a service. The request for this service is related to a required port, and the result of this service is associated to a provided port. Finally, attachments are defined to link the components to the connector itself (Fig. 15c lines 15–18).

Applying this mapping in our running example (Fig. 13), six components will be generated: *Smart home system* (the main actor); *Preference manager*, *Communication*, and *Data storage* (actors not related to the application domain); *Monitor* and *Actuator* (actors included to provide adaptability). Each dependency is mapped to a connector and the roles of their connectors (depender or dependee) will be defined according to the dependency direction. Furthermore, the roles of depender and dependee are mapped to connector roles that are comprised by the connector. In addition, required ports (where the actor is a depender) and provided ports (where the actor is a dependee) are defined. For instance, when an actor has at least one dependency as a dependee, its equivalent component will have at least one provided port. For instance, the *Monitor* component will have a provided port considering the *Environment monitored* dependency.

The dependencies on which the dependee is the *Actuator* actor all act on the environment. Therefore, in order to simplify our model, instead of creating one connector for each one of these dependencies, we propose the usage of a

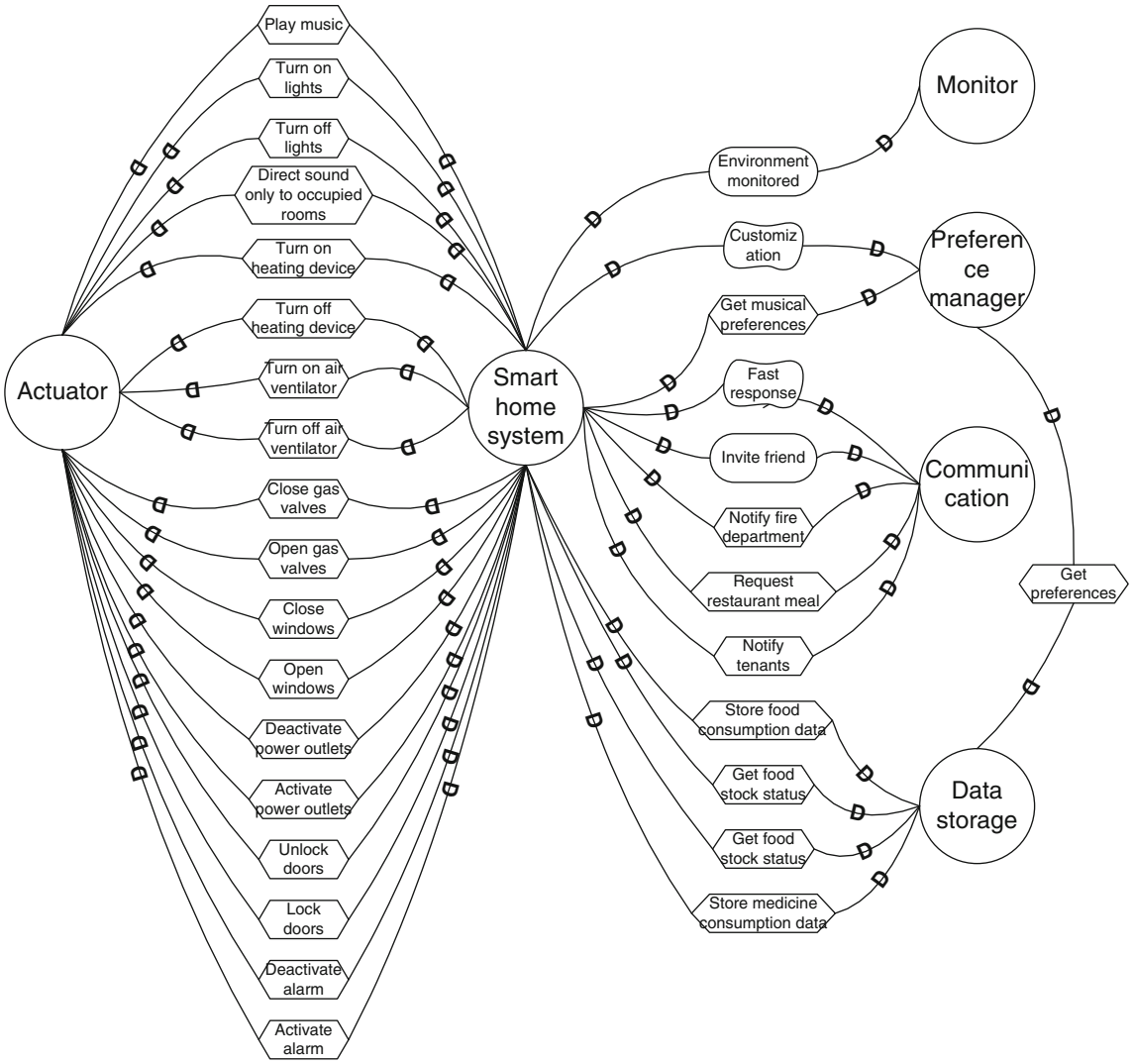


Fig. 13 SD model of the system after defining the *Actuator* and *Monitor* actors

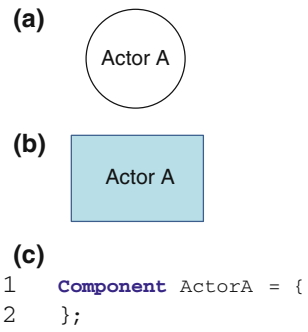


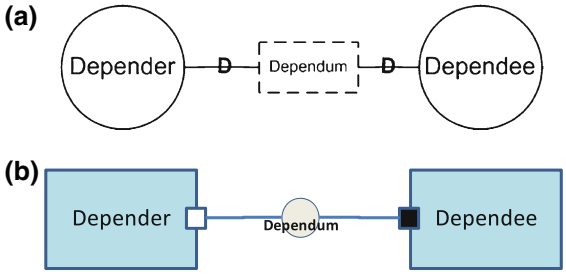
Fig. 14 Mapping an *i** actor onto an Acme component. **a** Source. **b** Target in graphical notation. **c** Target in textual notation

single *Actuations* connector. Later, this connector is refined to sub-connectors—one for each dependency.

Figure 16 shows an example of such a mapping, from an excerpt of our running example. The three dependencies are mapped as sub-connectors (Fig. 16c lines 16–28) of a single connector (Fig. 16c line 11). In Acme, a sub-element is defined as an element of a *system* inside a *representation* of the super element. The sub-elements are linked to the super elements through *bindings* (Fig. 16c lines 29–36).

Another particular case is that of dependencies with context annotations. These annotations, which specify a condition on which that dependency can be satisfied, will be mapped to a *pre-condition* property of the respective connector. An example of such a mapping is presented in Fig. 17.

Figure 18 presents the smart home system architectural model in Acme, generated from the application of these vertical transformation rules. This is an initial architectural model, yet to be refined in the following activities.



(c)

```
1   Component Depender = {
2       Port port1 = {
3           Property Required : boolean = true;
4       };
5   };
6   Component Dependee  = {
7       Port port2 = {
8           Property Provided : boolean = true;
9       };
10  };
11  Connector Dependum = {
12      Role dependerRole;
13      Role dependeeRole;
14  };
15  Attachments {
16      Depender.port1 to Dependum.dependerRole;
17      Dependee.port2 to Dependum.dependeeRole;
18  };
```

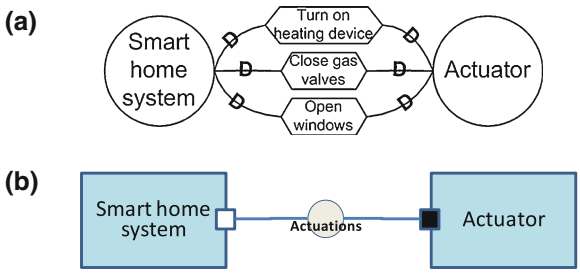
Fig. 15 Mapping an *i** generic dependency onto an Acme component. **a** Source. **b** Target in graphical notation. **c** Target in textual notation

5.2 Refine architectural model

From the initial architectural model with the main components of the system, defined in the previous activity, we are now going to define some of their sub-components. This will be performed based on the alternative sensor devices defined in the *Identification of sensors & actuators* activity. Recall that each data entity to be monitored was refined (see Fig. 12) to define the possible ways of monitoring it, as well as the impact of each alternative toward non-functional requirements expressed in softgoals.

At this point, further information may be required to provide more details and enable a better decision. For instance, in our example, it may help to compare different detector brands. Then, based on the priority of each softgoal, each alternative selection can be analyzed toward an optimal solution. The best solution can be automatically discovered using top–down algorithms [22]. Alternatively, one can make a selection of the devices and then analyze how this selection impacts the satisficing of the softgoals, using bottom–up algorithms [22].

For instance, if in the example of Fig. 12, the softgoal with higher priority is *Precision* then a device with ultrasonic detection will be selected. If instead the *Low cost* softgoal has higher priority, the device with passive infrared detection is selected. Nonetheless, a non-optimal set of devices (regarding the softgoals) may be selected due to another criteria, such as enhanced interoperability.



(c)

```
1  Component SmartHomeSystem = {
2    Port port1 = {
3       Property Required : boolean = true;
4    };
5  };
6  Component Actuator  = {
7    Port port2 = {
8       Property Provided : boolean = true;
9    };
10 };
11 Connector Actuations = {
12   Role dependerRole;
13   Role dependeeRole;
14   Representation {
15     System ActuationsDetails = {
16       Connector TurnOnHeatingDevice = {
17         Role dependerRole;
18         Role dependeeRole;
19       };
20       Connector CloseGasValves = {
21         Role dependerRole;
22         Role dependeeRole;
23       };
24       Connector OpenWindows = {
25         Role dependerRole;
26         Role dependeeRole;
27       };
28     };
29     Bindings {
30       dependerRole to
TurnOnHeatingDevice.dependerRole;
31       dependerRole to CloseGasValves.dependerRole;
32       dependerRole to OpenWindows.dependerRole;
33       dependeeRole to
TurnOnHeatingDevice.dependeeRole;
34       dependeeRole to CloseGasValves.dependeeRole;
35       dependeeRole to OpenWindows.dependeeRole;
36     };
37   };
38 };
39 Attachments {
40   SmartHomeSystem.port1 to Actuations.dependerRole;
41   Actuator.port2 to Actuations.dependeeRole;
42 };
```

Fig. 16 Example of the specific mapping of the dependencies on which the Actuator is a dependee. **a** Source. **b** Target in graphical notation. **c** Target in textual notation

Once defined which are the devices to be used they will be mapped onto Acme components, using the first vertical transformation rule. However, instead of being components of the overall system, they are sub-components of the *Monitor* component. Figure 19 exemplifies the specification of two sub-components of the *Monitor* component.

In Acme, sub-components are defined as components of a (sub) *system* inside a *representation* of the component (Fig. 19b lines 5–17). The sub-components are linked to the external port of the *Monitor* component through *bindings* (Fig. 19c lines 18–21).

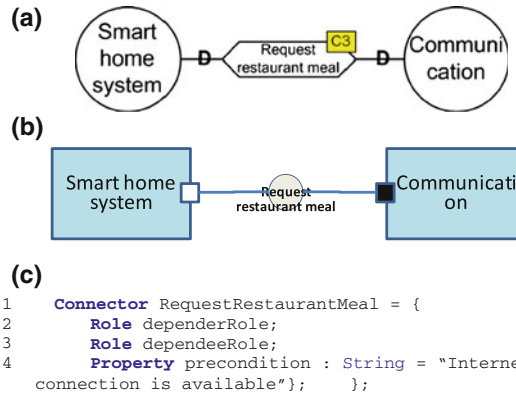


Fig. 17 Example of the specific mapping of dependencies on which there is a contextual annotation. **a** Source. **b** Target in graphical notation. **c** Target in textual notation

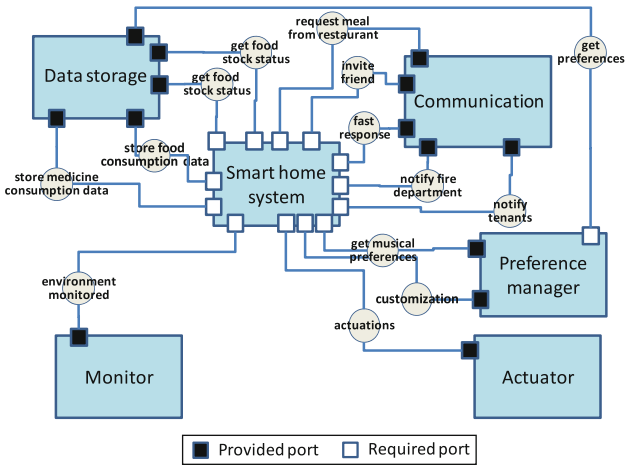


Fig. 18 The result of mapping the smart home system model from *i** to Acme

A similar refinement will be performed for the *Actuator* component. In the *Identification of sensors & actuators* activity, the tasks of the *Actuator* actor were defined. Now, these tasks will be used as input to discover the sub-components of the *Actuator* component. This is achieved by analyzing each one of its tasks, identifying what is the real-world entity that will be changed by that task and defining an actuator for it. For instance, the *Turn on lights* and *Turn off lights* will affect the lights of the house, requiring a lights actuator. The *Open windows* and *Close windows* tasks affect the windows of the house, requiring a windows actuator, and so on. Each actuator will be a sub-component of the *Actuator* component. These sub-components will be linked to the external port of *Actuator* through a connector that simply pass forward the information about the required actuation.

The results of this activity on the smart home system are presented in Fig. 20. It shows the *Monitor* and the *Actuator* components, linked to the main component, as well as their sub-components.

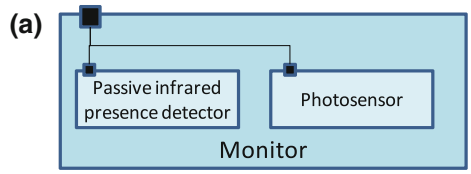
5.3 Include the self-adaptation component

In this activity, the component responsible for the self-adaptation reasoning [16] will be incorporated into the architecture. These components perform a Monitor-Diagnose-Compensate (MDC) reasoning cycle, as introduced in [17], to check whether the goals of the system are being achieved and, if not, what adaptations are required to achieve them. Figure 21 shows the smart home system architecture already with the *Self-adaptation* component. The *Self-adaptation* component will be linked to the main component of the system (in this example, the *Smart home system* component), to the *Monitor* component and to the *Actuator* component. The *Self-adaptation* component will receive a history of the system's execution from the main component (*log* connector) and the environmental data from the *Monitor* component (*environment Monitored* connector). These data will be checked against the goal model of the system, and the required adaptations will be identified. Some of the adaptations will be required to be performed through the *Actuator* component (*actuations* connector) and others will be suggested to the main component (*system pushes* connector).

Figure 22 presents the architecture of the *Self-adaptation* component in more details. There is a sub-component for each one of the MDC steps, plus a *Policy manager*. The *Monitor* sub-component of the self-adaptation component receives data about changes in the context and the system execution through connectors to the *Monitor* and to the main component of the system. These data, gathered from different sensors, will be normalized by the *Event normalizer* sub-component. For example, if two different temperature sensors provide the current temperature, respectively, on Fahrenheit and Celsius scales, these data will be normalized to use a single scale. The normalized data will be used by the *Dependency monitor* to assess the status of the dependencies, by the *Context monitor* to update the context data, and by the *Task execution monitor* to identify whether the tasks of the system were successfully performed. This information will be passed to the *Diagnoser* sub-component through the *Dependencies status* connector, the *Current context* connector, and the *Task execution status* connector, respectively.

The *Diagnoser* sub-component will use these data to identify failures in the system execution, based on the annotated goal model. The failures may concern unachieved goals, unsatisfied dependencies, and tasks that were not performed (and should have been). The *Contextual goal model manager* uses the information of the current context to check in the goal model that goals and tasks should, can, and cannot be achieved. These data will be provided to the *Task execution diagnoser*, the *Dependency diagnoser*, and the *Goal commitment diagnoser*, through

Fig. 19 Example of a connection between the *Monitor* and two of its sub-components. **a** Graphical notation. **b** Textual notation



(b)

```
1   Component Monitor = {
2     Port port1 = {
3       Property Required : boolean = true;
4     };
5     Representation {
6       System MonitorDetails = {
7         Component PassiveInfraredPresenceDetector = {
8           Port p = {
9             Property Provided : boolean = true;
10          };
11        };
12        Component Photosensor = {
13          Port p = {
14            Property Provided : boolean = true;
15          };
16        };
17      };
18      Bindings {
19        port1 to PassiveInfraredPresenceDetector.p;
20        port1 to Photosensor.p;
21      };
22    };
23  };
```

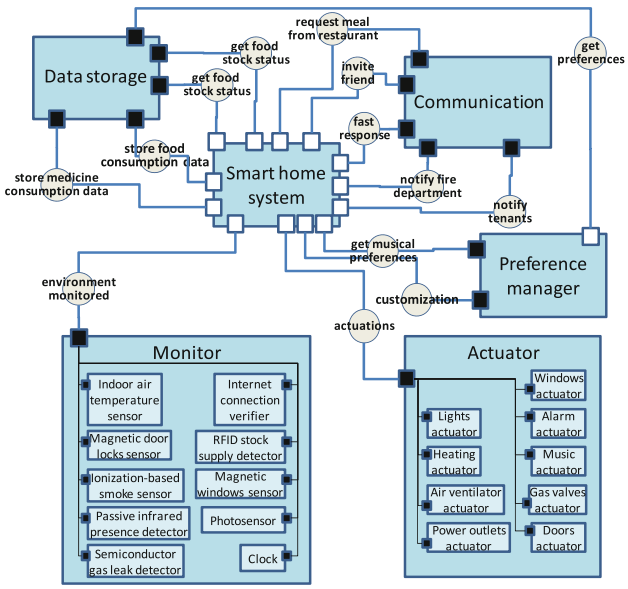


Fig. 20 The smart home system architecture after defining the sub-components of the *Monitor* and *Actuator* components

the *Goals/tasks applicability* connector. Each one of these sub–sub-components will identify the *Failed dependencies*, the *Failed tasks/goals*, and the *Uncommitted goals*, respectively. An uncommitted goal is a goal that has not failed yet but which no task has been initiated to achieve it. All this failure information will be consolidated as a list of failures on the *Failure diagnoser*, based on the *Tolerance policies* provided by the *Policy manager*. The *Policy manager* is responsible for providing a list of tolerance rules that specify whether a failure should be ignored or not on some conditions [41]. These rules are expressed on a policy defined by the system administrator or by the user herself. Based on this policy, the *Failure diagnoser* will discard the failures for which the rules apply and provide the list of failures to the *Compensator* sub-component, through the *Failure diagnosis* connector.

In the *Compensator* sub-component, the *Prioritize diagnosis* will prioritize the list of failures to be compensated (i.e., that will require an adaptation) based on *Priority policies* also provided by the *Policy manager*. The failures with higher priority will be provided to the *Reaction strategy selector* through the *Selected Diagnosis* connector. The *Reaction strategy selector* will define which compensation should be performed to address each failure. The compensation, or adaptation, may be a change on the system itself (*Push system compensations*) or on its environment (*Actuate compensations*). The *System pushing* will suggest the push system compensations to the main component of the system (*system pushes* connector on Fig. 21), and the *Actuator manager* will require the actuation compensations to be performed by the *Actuator* component (*actuations* connector on Fig. 21).

Therefore, the *Self-adaptation* component is responsible for performing the reasoning related to adaptation, but it is not responsible for performing the adaptation itself. The

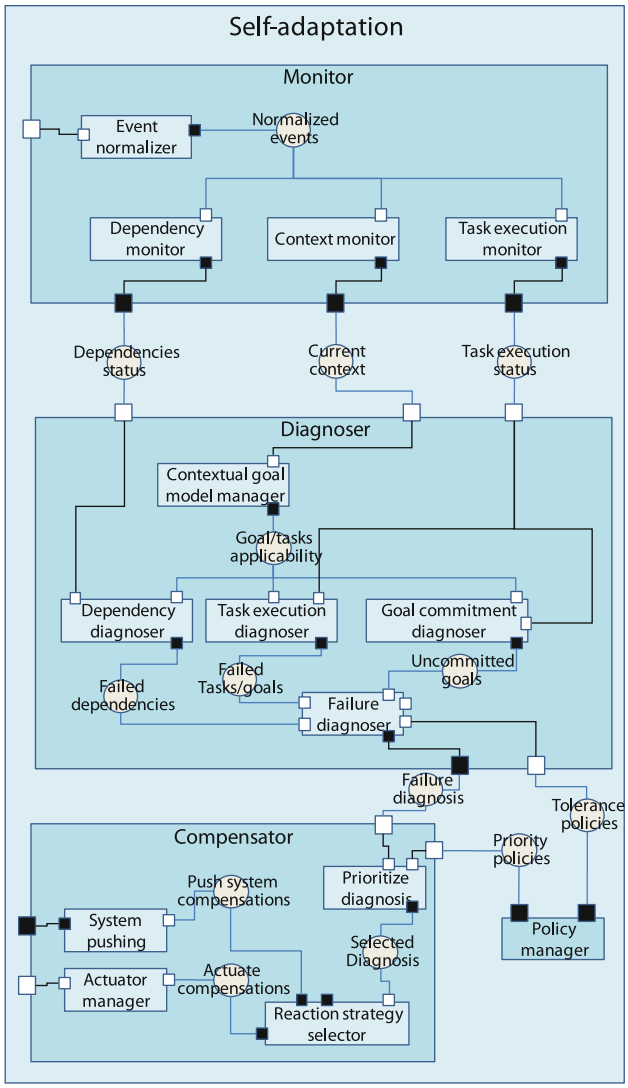


Fig. 21 Architectural model presenting all the internal components of the *Self-adaptation* component. Adapted from [16]

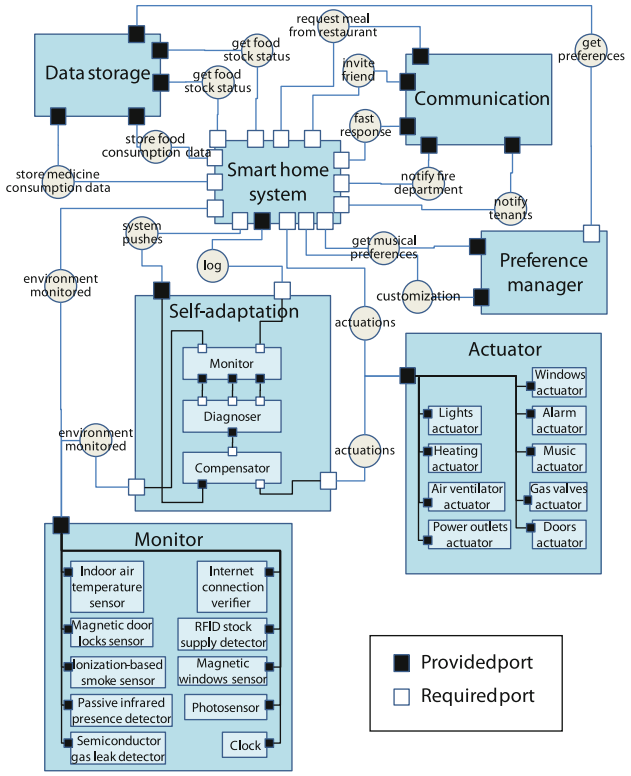


Fig. 22 The final architecture for the smart home system, including the *Self-adaptation* component

Monitor component provides data that allow the *Self-adaptation* component to check whether the goals of the system are being achieved, accordingly to the context-annotated goal model. Once identified whether and which adaptations will be required, the *Self-adaptation* component then requests the *Actuator* component to perform the adaptations themselves. More details on the algorithms used by the *Self-adaptation* component can be found in [16].

Without the *Self-adaptation* component, this MDC cycle would need to be hard-coded and repeated in several different parts of the system's source code. This would increase the cost for performing changes on the system, make more difficult the analysis of the adaptive behavior, and possibly increase the occurrence of errors. By encapsulating all this reasoning on a single component, all the adaptation reasoning will be able to be performed with a high level of abstraction and therefore easy to evolve, without requiring any change on the system's source code. So, if the adaptive behavior of the system need to be changed, only the enriched goal model will need to be changed—unless this change requires a new kind of monitoring or actuation. Moreover, since the *Self-adaptation* component is domain-independent, it can be reused in different systems. On the other hand, such generality might provoke some loss in performance. Further studies are required in order to identify the degree of this loss.

It is important to note that not all events will result in compensation actions. Just the interactions that were captured by the monitor component, diagnosed as failures and filtered by the police manager can receive compensation actions.

6 Discussion

Goal-oriented approaches have long been proposed for understanding and reasoning on requirements [31]. We claim that they are also appropriate for reasoning on and developing architectures. They offer a unified framework in which both functional and non-functional concerns can be integrated, and refinement/abstractions links are

precisely defined and provide the basis for various forms of qualitative, quantitative, or formal reasoning [31]. Note that others have also discussed the application of model transformations between *i** based models. For example, the work presented in [9] 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.

The original Strategy for Transition between Requirements models and Architectural Models (STREAM) [34] is a generic process aimed to define modular architectures with a model-driven approach. It strongly relies on transformation rules to incrementally evolve a requirements model in *i** and then derive architecture models. The resulting requirements model is closer to an earlier architectural design, facilitating the transition from requirements to architectural design. From the modular *i** model, an Acme architectural solution is derived through a set of mappings between the concepts of *i** and Acme languages. Afterward, one of the architectural solutions is chosen according to NFRs established in the requirements phase. If necessary, this architecture is further detailed through refinement patterns [30]. These patterns are chosen and applied. Therefore, these horizontal and vertical transformations involve steps before and after the change of notation. For example, in the transformation rules of *i**, there often occurs division of actors, while in the activity of refinement of the architecture can occur grouping of components. However, it does not address the specific problem of developing adaptive systems.

In comparison with the original STREAM, in our approach, we address the development of adaptive systems by including three new activities: *Context Annotation and Analysis*, *Identification of Sensors and Actuators*, *Include the Self-Adaptation Component*. In addition, the activity *Refine Architectural Model* was modified to refine with more details the *Monitor* and *Actuator* components. The vertical transformation rules of *Generate Architectural Model* were complemented to support context annotations on dependencies. Lastly, for *Requirements Refactoring*, we defined an improved set of horizontal transformation rules, contemplating some cases that were missing.

The proposed approach might not be suitable for all systems and usage scenarios. It is worth noting that our approach generates architecture in which the adaptation reasoning is centralized in a single component. Hence, it might not be suitable for systems that require high scalability or performance. Further studies are required in order to identify on which scenarios the STREAM-A approach is more suitable.

In the following subsections, we discuss our approach in comparison with other approaches for architecture derivation and for development of adaptive systems.

6.1 Architecture derivation from goal models

We also discuss some approaches that produce architectural design considering goal models as source models: *i** [5], KAOS [30], and AOV-graph [43]. The SIRA approach [5] 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 [12]. Both requirements and architecture models are described using the *i** language. An organizational architectural style is chosen based on a catalog of non-functional requirements presented in [29]. *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 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 Acme elements to reach early architectural design solutions. Furthermore, our approach is focused on the development of adaptive systems.

Lamsweerde [30] defines a method to produce architectural models from KAOS requirements models. In that 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 consists of a graphical tree and a formal language. In our approach, we use another goal model language as input, the *i** model. In fact, we advocate that first we need to modularize the *i** models by means of horizontal transformations. 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. Then again, our approach is specifically designed to address the development of adaptive systems.

In [43], a set of mapping rules is proposed between the Aspectual oriented V-graph (AOV-graph) and the

AspectualACME, an ADL based in Acme. This approach does not address the adaptability softgoal. 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, this traceability is implicitly defined by preserving the names of the elements—actors and components; dependencies and connectors—on the one-to-one transformations. However, we acknowledge that this is not enough to provide robust traceability and evolution control. Another approach that handles aspectual goal models is presented in [1]. It enables the expression of more modular goal models through the definition of crosscutting concerns. However, it does not tackle the derivation of architectures from the goal model.

In the context of this work, we are using *i** models to produce architectural models in Acme [21]. 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 tend to be complex. The first activity of the STREAM-A process is a first step to address this issue.

The *i** modeling language is rich and expressive in describing the system requirements [47]. The approaches that use *i** modeling language as the starting point of software specification, such as RISD [25], Tropos [12], and PRIM [26], do not support a systematic transition from requirements specifications to architectural design description. Architects often 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.

6.2 Adaptive approaches based on goal models

There are several approaches for developing adaptive systems based on goal models. In this subsection, we are going to discuss some of them, including those on which our process is based.

Lapouchnian and Mylopoulos [33] and Ali et al. [2] use the notion of context to express domain variability. The goal model is annotated with context expressions that define conditions on the model elements. During runtime, a system may check whether a task being performed is allowed on that context and, if not, it may change its behavior. Both approaches are concerned with reasoning at requirements level, without prescribing any specific architecture. The approach of Ali et al. [2] is used in our process to define the context annotations and models (see Sect. 4.2).

Dalpiaz et al. [16] also uses context-enriched goal models, aiming to deploy adaptive systems. Besides constraining the selection of alternatives, the context is used to define activation events and commitment conditions for goals and preconditions to tasks. Compensations are also defined to mitigate the occurrence of failures. Their approach describes the architecture of a component responsible for performing the adaptation-related reasoning. However, it does not prescribe how to define the architecture of the system that will interact with this component. Throughout the STREAM-A process, the requirements model of a system is enriched to include all the information necessary for the adaptation reasoning that will be performed by the component (Context Annotation and Analysis; Identification of Sensors and Actuators), the architecture for the system is derived (Generate Architecture Model; Refine Architectural Model), and then the self-adaptation component is integrated onto the architecture (Include the Self-Adaptation Component).

Morandini, Penserini, and Perini [38] propose to use goal models enriched with environmental and fault modeling. The goals status is expressed in terms of environment conditions, similar to the context annotations. Fault modeling is used to define situations on which recovery activities may be performed to prevent or mitigate a fault. This is similar to the concept of obstacles that is part of KAOS. Besides being a comprehensive approach, it is only suited to develop multi-agent systems.

At another level of requirements engineering for adaptive systems [7], there are some approaches based on the notion that requirements might change at runtime and that the system should be able to respond to these changes with minimal human intervention. However, as of today, there are still too many open issues on these approaches, such as how to express the new requirements in a way that is both simple to the user to define and that can be understood by the machine. Nonetheless, the STREAM-A process could be further extended, in the future, to consider the new developments originated from promising approaches, such as the ones described below.

Jian et al. [27] allows the insertion of goals at runtime. However, to respond to these changes, new modules must be incorporated to the system as well. This approach also uses a notation for expressing environmental conditions similar to the contextual approaches above.

Qureshi et al. [42] also allows the changing of goal models at runtime: add goal, add means-end, suspend

means-end, resume means-end, and relax means-end. To address these news goals, it uses a service-based architecture, on which a lookup mechanism will identify services that may satisfy the new requirements. The services may either already exist on the system's pool or may be found through web service search mechanisms.

Bencomo et al. [6] also deals with the notion of changing goal models at runtime, through requirements reflection. Additionally, it uses a flexibility language to deal with uncertainty.

Lastly, Baresi, and Pasquale [3] propose the use of adaptive goals, in contrast to conventional goals. The adaptive goals specify countermeasures to be performed when a conventional goal is violated. The countermeasures may involve the changing of the model at runtime, but this is fundamentally different from the other approaches since these changes are defined at design time.

7 Conclusion and future work

This paper presented STREAM-A, a process to generate an architectural model addressing the adaptability requirement. The first activity prepares the requirements model to balance the responsibilities of a system actor, delegating them to other new system actors. It consists of a set of horizontal transformation rules defined to support the refactoring of goal models. Moreover, context annotations are inserted in the goal model to specify which environmental changes should be monitored and how they affect the system behavior. In the third activity, the sensors and actuators of the system are defined.

From this close to architecture goal model, the fourth activity of the STREAM process derives an early architectural model described in Acme. Then, the architecture is refined with the sub-components that represent the sensor and actuator devices. The selection of the sensor devices is based on the non-functional requirements defined in the requirements model. Finally, in the last activity, a specific self-adaptation component is included in the architecture. This allows the handling of adaptability in a high abstraction level, without the need of coding the adaptation behavior.

As future work, we expect to develop tool support for our approach. Besides the usual goal modeling and Acme modeling, such a tool would need to allow context annotation on the goal models and to implement both the horizontal and vertical transformation rules here defined. Such tool support would allow us to investigate the scalability of our approach in some real life complex projects.

Our approach still needs to be improved to support other architectural views. In the future, we intent to support architectural models required to deal with behaviors of the architecture and technological issues.

Traceability mechanisms will also be defined. This would make it easier to maintain the consistency between the requirements and the architecture models throughout the system evolution. We are also interested in investigating if the STREAM process could be extended to address other non-functional requirements and their related approaches—if any. Such non-functional requirements of interest include security and accessibility.

Finally, we acknowledge that a thorough experimentation must be performed in order to evaluate and improve the STREAM-A process. Such an experiment is being defined using the framework proposed by Wohlin et al. [46] for performing experiments in software engineering and the Architecture Tradeoff Analysis Method (ATAM) [4] for evaluating the resulting architecture.

Acknowledgments This work has been supported by the Brazilian institutions National Council for Scientific and Technological Development (CNPq), Coordination for the Perfecting of High Education Personnel (CAPES), and by the Erasmus Mundus External Cooperation Window - Lot 15 Brazil.

References

1. Alencar F, Castro J, Lucena M, Santos E, Silva C, Araújo J, Moreira A (2010) Towards modular *i** models. In: Proceedings of the SAC'10 proceedings of the 2010 ACM symposium on applied computing. New York, USA, pp 292–297
2. Ali R, Dalpiaz F, Giorgini P (2010) A goal-based framework for contextual requirements modeling and analysis. Req Eng J 15(4):439–458. doi:10.1007/s00766-010-0110-z
3. Baresi L, Pasquale L (2010) Live goals for adaptive service compositions. In: Proceedings of the 2010 ICSE workshop on software engineering for adaptive and self-managing systems (SEAMS'10). New York, USA, pp 114–123. doi:10.1145/1808984.1808997
4. Bass L, Clements P, Kazman R (2003) Software architecture in practice. Wesley Longman Publishing Co, Addison
5. Bastos LRD, Castro J (2005) From requirements to multi-agent architecture using organisational concepts. SIGSOFT Softw Eng Notes 30(4):1–7. doi:10.1145/1082983.1082980
6. Bencomo N, Whittle J, Sawyer P, Filkenstein A, Letier E (2010) Requirements Reflection—requirements as runtime entities. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering, vol 2 (ICSE'10). New York, USA, pp 199–202. doi:10.1145/1810295.1810329
7. Berry DM, Cheng B, Zhang J (2005) The four levels of requirements engineering for and in dynamic adaptive systems. In: Proceedings of the 11th international workshop on requirements engineering foundation for software quality (REFSQ). Porto, Portugal, p 5
8. Boer RCD, Vliet HV (2009) On the similarity between requirements and architecture. J Syst Softw 82(3):544–550. doi:10.1016/j.jss.2008.11.185
9. Bresciani P, Perini A, Giorgini P, Giunchiglia F, Mylopoulos J (2002) Modeling early requirements in Tropos: a transformation based approach. Agent-Oriented Software Engineering II—LNCS 2222/2002:151–168. doi:10.1007/3-540-70657-7_11

10. Castro J, Franch X, Mylopoulos J, Yu E (2010) Fourth international *i** Workshop (i* '10). CEUR workshop proceedings, Hammamet, Tunisia, p 586
11. Castro J, Kolp M, Liu L, Perini A (2009) Dealing with complexity using conceptual models based on Tropos. Conceptual modeling: foundations and applications—essays in honor of John Mylopoulos. LNCS 5600:335–362. doi:10.1007/978-3-642-02463-4_18
12. Castro J, Kolp M, Mylopoulos J (2002) Towards requirements-driven information systems engineering the Tropos project. Inf Syst 27(6):365–389. doi:10.1016/S0306-4379(02)00012-1
13. Castro J, Silva C, Mylopoulos J (2003) Modeling organizational architectural styles in UML. Adv Inf Syst Eng LNCS 2681: 111–126. doi:10.1007/3-540-45017-3_10
14. Chung L, Nixon BA, Yu E, Mylopoulos J (2000) Non-functional requirements in software engineering. Springer, Berlin
15. Czarnecki K, Helsen S (2003) Classification of model transformation approaches. In: Proceedings of the 2nd workshop on generative techniques in the context of model-driven architecture. Anaheim, USA
16. Dalpiaz F, Giorgini P, Mylopoulos J (2009) An architecture for requirements-driven self-reconfiguration. Adv Inf Syst Eng LNCS 5565:246–260. doi:10.1007/978-3-642-02144-2_22
17. Dalpiaz F, Giorgini P, Mylopoulos. J Software Self-Reconfiguration: a BDI-based approach (Extended Abstract). Knowledge creation diffusion utilization (Algorithm 1)
18. Dardenne A, Lamsweerde A, van Fickas S (1993) Goal-directed requirements acquisition. Sci Comput Prog 20(1–2):3–50. doi:10.1016/0167-6423(93)90021-G
19. Dijkstra E (1975) A discipline of programming. Prentice-Hall, Englewood Cliffs
20. Estrada H, Rebollar AM, Pastor O, Mylopoulos J (2006) An empirical evaluation of the *i** framework in a model-based software generation environment. Adv Inf Syst Eng LNCS 4001: 513–527. doi:10.1007/11767138_34
21. Garlan D, Monroe R, Wile D (1997) Acme: an architecture description interchange language. In: Proceedings of the 1997 conference of the centre for advanced studies on collaborative research (CASCON'97). Toronto, Canada
22. Giorgini P, Mylopoulos J, Nicchiarelli E, Sebastiani R (2003) Formal reasoning techniques for goal models. J Data Semant LNCS 2800:1–20. doi:10.1007/978-3-540-39733-5_1
23. Goulão M, Abreu FBE (2003) Bridging the gap between Acme and UML 2.0 for CBD. In: Proceedings of the specification and verification of component-based systems—SAVCBS 2003. Helsinki, pp 75–89
24. Grau G, Franch X (2007) On the adequacy of *i** models for representing and analyzing software architectures. Advances in conceptual modeling: foundations and applications. LNCS 4802:296–305
25. Grau G, Franch X, Mayol E, Ayala C, Cares C, Haya M, Navarrete F, Botella P, Quer C (2005) RiSD: A methodology for building *i** strategic dependency models. In: Proceedings of the 17th international conference on software engineering and knowledge engineering (SEKE'05). Taipei, Taiwan, pp 259–266
26. Grau G, Franch X, Ávila S (2006) J-PRiM: A java tool for a process reengineering *i** methodology. In: Proceedings of the 14th IEEE international conference on requirements engineering (RE'06). Minneapolis, USA, pp 359–360. doi:10.1109/RE.2006.36
27. Jian Y, Li T, Liu L, Yu E (2010) Goal-oriented requirements modelling for running systems. In: Proceedings of the 1st international workshop on requirements@run-time. Sidney, Australia
28. Jureta IJ, Borgida A, Ernst NA, Mylopoulos J (2010) Techne: Towards a new generation of requirements modeling languages with goals, preferences, and inconsistency handling. In: Proceedings of the 18th IEEE international requirements engineering conference (RE'10)
29. Kolp M, Giorgini P, Mylopoulos J (2006) Multi-agent architectures as organizational structures. Auto Agents Multi-Agent Syst 13(1):3–25. doi:10.1007/s10458-006-5717-6
30. Lamsweerde AV (2003) From system goals to software architecture. Formal methods for software architectures. LNCS 2804:25–43. doi:10.1007/978-3-540-39800-4_2
31. Lamsweerde AV (2001) Goal-oriented requirements engineering a guided tour. In: Proceedings of the 5th IEEE international symposium on requirements engineering. Toronto, Canada, pp 249–262. doi:10.1109/ISRE.2001.948567
32. Lamsweerde AV (2004) Goal-oriented requirements enginering: a roundtrip from research to practice. In: Proceedings of the 12th IEEE international requirements engineering conference (RE 2004). Kyoto, Japan, pp 4–7
33. Lapouchnian A, Mylopoulos J (2009) Modeling domain variability in requirements engineering with contexts. Conceptual Modeling—ER 2009. LNCS 5829:115–130. doi:10.1007/978-3-642-04840-1_11
34. Lucena M (2010) STREAM: a systematic process to derive architectural models from requirements models. Thesis, Universidade Federal de Pernambuco
35. Lucena M, Castro J, Silva C, Alencar F, Santos E, Pimentel J (2009) A model transformation approach to derive architectural models from goal-oriented requirements models. On the move to meaningful internet systems: OTM 2009 workshops. LNCS 5872:370–380. doi:10.1007/978-3-642-05290-3_49
36. Lucena M, Silva C, Santos E, Alencar F, Castro J (2009) Applying transformation rules to improve *i** models. In: Proceedings of the 21st international conference on software engineering and knowledge engineering (SEKE09). Boston, USA, pp 43–48
37. Mens T, Czarnecki K, Van Gorp P (2005) A taxonomy of model transformations. In: Proceedings of the language engineering for model-driven software development. Dagstuhl, Germany
38. Morandini M, Penserini L, Perini A (2008) Towards goal-oriented development of self-adaptive systems. In: Proceedings of the 2008 ICSE workshop on software engineering for adaptive and self-managing systems (SEAMS'08). New York, USA, pp 9–16. doi:10.1145/1370018.1370021
39. Others (2009) Software engineering for self-adaptive systems: a research roadmap. Software engineering for self-adaptive systems. LNCS 5525/2009:1-26. doi:10.1007/978-3-642-02161-9_1
40. Pastor O, Molina JC (2010) Model-driven architecture in practice: a software production environment based on conceptual modeling. Springer, Berlin
41. Pimentel J, Santos E, Castro J (2010) Conditions for ignoring failures based on a requirements model. In: Proceedings of the 22nd international conference on software engineering and knowledge engineering—SEKE 2010. Redwood, USA, pp 48–53
42. Qureshi NA, Perini A, Ernst NA, Mylopoulos J (2010) Towards a continuous requirements engineering framework for self-adaptive systems. In: Proceedings of the 1st international workshop on requirements@run-time. Sidney, Australia
43. Silva LF, Batista TV, Garcia A, Medeiros AL, Minora L (2007) On the symbiosis of aspect-oriented requirements and architectural descriptions. Early aspects: current challenges and future directions. LNCS 4765:75–93. doi:10.1007/978-3-540-76811-1_5
44. Subramanian N, Chung L (2001) Software architecture adaptability: An NFR approach. In: Proceedings of the 4th international workshop on principles of software evolution (IWPSE'01). New York, USA, pp 52–61. doi:10.1145/602461.602470
45. Taylor RN, Medvidovic N, Dashofy E (2009) Software architecture: foundations, theory and practice. Wiley, New York

46. Wohlin C, Runeson P, Höst M, Regnell B, Wesslen A (2000) Experimentation in software engineering: an introduction. Kluwer Academic Publishers, The Netherland
47. Yu E (1995) Modelling strategic relationships for process reengineering. Thesis, University of Toronto
48. Yu E, Castro J, Perini A (2008) Strategic actors modeling with *i** Tutorial notes. In: Proceedings of the 16th IEEE international requirements engineering conference (RE'08). Barcelona, Spain
49. Yu E, Mylopoulos J (1998) Why goal-oriented requirements engineering. In: Proceedings of the 4th international workshop on requirements engineering: foundations of software quality—REFSQ'98. Pisa, Italy, pp 15–22
50. Yu Y, Leite JCSDP, Mylopoulos J (2004) From goals to aspects: discovering aspects from requirements goal models. In: Proceedings of the 12th IEEE international requirements engineering conference (RE 2004). Kyoto, Japan, pp 33–42. doi:10.1109/ICRE.2004.1335662